Operational notes

Document updated on **June 13, 2022**.

The following colors are **not** part of the final product, but serve as highlights in the editing/review process:

- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- terms that have not yet been defined in the book
- things that need to be checked only at the very final typesetting stage
  (and it doesn't make sense to do them before)
- text that needs advice from the communications/marketing team: Aaron & Shane
- text that needs to be completed or otherwise edited (by Sylvia)

Todo list

MoonMath manual

TechnoBob and the Least Scruples crew

June 13, 2022

Contents

Chapter 1

Introduction

This is dump from other papers as inspiration for the intro:

Zero knowledge proofs are a class of cryptographic protocols in which one can prove honest computation without revealing the inputs to that computation. A simple high-level example of a zero-knowledge proof is the ability to prove one is of legal voting age without revealing the respective age. In a typical zero knowledge proof system, there are two participants: a prover and a verifier. A prover will present a mathematical proof of computation to a verifier to prove honest computation. The verifier will then confirm whether the prover has performed honest computation based on predefined methods. Zero knowledge proofs are of particular interest to public blockchain activities as the verifier can be codified in smart contracts as opposed to trusted parties or third-party intermediaries.

Zero-knowledge proofs (ZKPs) are an important privacy-enhancing tool from cryptography. Theyallow proving the veracity of a statement, related to confidential data, without revealing any in-formation beyond the validity of the statement. ZKPs were initially developed by the aca-demiccommunity in the 1980s, and have seen tremendous improvements since then. They are now ofpractical feasibility in multiple domains of interest to the industry, and to a large commu-nity ofdevelopers and researchers. ZKPs can have a positive impact in industries, agencies, and for per-sonal use, by allowing privacy-preserving applications where designated private data can be madeuseful to third parties, despite not being disclosed to them.

ZKP systems involve at least two parties: a prover and a verifier. The goal of the prover is toconvince the verifier that a statement is true, without revealing any additional information. Forexample, suppose the prover holds a birth certificate digitally signed by an authority. In orderto access some service, the prover may have to prove being at least 18 years old, that is, thatthere exists a birth certificate, tied to the identify of the prover and digitally signed by a trustedcertification authority, stating a birthdate consistent with the age claim. A ZKP allows this, withoutthe prover having to reveal the birthdate.

1.1 Target audience

This book is accessible for both beginners and experienced developers alike. Concepts are gradually introduced in a logical and steady pace. Nonetheless, the chapters lend themselves rather well to being read in a different order. More experienced developers might get the most benefit by jumping to the chapters that interest them most. If you like to learn by example, then you should go straight to the chapter on Using Clarinet.

It is assumed that you have a basic understanding of programming and the underlying logical concepts. The first chapter covers the general syntax of Clarity but it does not delve into what programming itself is all about. If this is what you are looking for, then you might have a more difficult time working through this book unless you have an (undiscovered) natural affinity for such topics. Do not let that dissuade you though, find an introductory programming book and press on! The straightforward design of Clarity makes it a great first language to pick up.

1.2 The Zoo of Zero-Knowledge Proofs

First, a list of zero-knowledge proof systems:

1. Pinocchio (2013): Paper
   - Notes: trusted setup
2. BCGTV (2013): Paper
   - Notes: trusted setup, implementation
3. BCTV (2013): Paper
   - Notes: trusted setup, implementation
4. Groth16 (2016): Paper
   - Notes: trusted setup
   - Other resources: Talk in 2019 by Georgios Konstantopoulos
5. GM17 (207): Paper
   - Notes: trusted setup
   - Other resources: later Simulation extractability in ROM, 2018
6. Bulletproofs (2017): Paper
   - Notes: no trusted setup
   - Other resources: Polynomial Commitment Scheme on DL, 2016 and KZG10, Polynomial Commitment Scheme on Pairings, 2010
7. Ligero (2017): Paper
   - Notes: no trusted setup
   - Other resources:
8. Hyrax (2017): Paper
   - Notes: no trusted setup
   - Other resources:
9. STARKs (2018): Paper
   - Notes: no trusted setup
   - Other resources:

10. Aurora (2018): Paper
    - Notes: transparent SNARK
    - Other resources:
11. Sonic (2019): Paper
    - Notes: SNORK - SNARK with universal and updateable trusted setup, PCS-based
    - Other resources: Blog post by Mary Maller from 2019 and work on updateable and universal setup from 2018
12. Libra (2019): Paper
    - Notes: trusted setup
    - Other resources:
13. Spartan (2019): Paper
    - Notes: transparent SNARK
    - Other resources:
14. PLONK (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources: Discussion on Plonk systems and Awesome Plonk list
15. Halo (2019): Paper
    - Notes: no trusted setup, PCS-based, recursive
    - Other resources:
16. Marlin (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources: Rust Github
17. Fractal (2019): Paper
    - Notes: Recursive, transparent SNARK
    - Other resources:
18. SuperSonic (2019): Paper
    - Notes: transparent SNARK, PCS-based
    - Other resources: Attack on DARK compiler in 2021
19. Redshift (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources:

Other resources on the zoo: Awesome ZKP list on Github, ZKP community with the reference document

To Do List

- Make table for prover time, verifier time, and proof size
- Think of categories - *Achieved Goals*: Trusted setup or not, Post-quantum or not, ...
- Think of categories - *Mathematical background*: Polynomial commitment scheme, ...
- ... while we discuss the points above, we should also discuss a common notation/language for all these things. (E.g. transparent SNARK/no trusted setup/STARK)

Points to cover while writing

- Make a historical overview over the "discovery" of the different ZKP systems
- Make reader understand what paper is build on what result etc. - the tree of publications!
- Make reader understand the different terminology, e.g. SNARK/SNORK/STARK, PCS, R1CS, updateable, universal, ...
- Make reader understand the mathematical assumptions - and what this means for the zoo.
- Where will the development/evolution go? What are bottlenecks?

Other topics I fell into while compiling this list

- Vector commitments: `https://eprint.iacr.org/2020/527.pdf`
- Snarkl: `http://ace.cs.ohio.edu/~gstewart/papers/snaarkl.pdf`
- Virgo?: `https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F19/projects/reports/project5_report_ver2.pdf`

Chapter 2

Preliminaries

2.1 Preface and Acknowledgements

This book began as a set of lecture and notes accompanying the zk-Summit 0x and 0xx .... It arose from the desire to collect the scattered information of snarks [] and present them to an audience that does not have a strong backgroud in cryptography []

2.2 Purpose of the book

The first version of this book is written by security auditors at Least Authority where we audited quite a few snark based systems. Its included "what we have learned" destilate of the time we spend on various audits.

We intend to let illus- trative examples drive the discussion and present the key concepts of pairing computation with as little machinery as possible. For those that are fresh to pairing-based cryptography, it is our hope that this chapter might be particu- larly useful as a first read and prelude to more complete or advanced expositions (e.g. the related chapters in [Gal12]).

On the other hand, we also hope our beginner-friendly intentions do not leave any sophisticated readers dissatisfied by a lack of formality or generality, so in cases where our discussion does sacrifice completeness, we will at least endeavour to point to where a more thorough exposition can be found.

One advantage of writing a survey on pairing computation in 2012 is that, after more than a decade of intense and fast-paced research by mathematicians and cryptographers around the globe, the field is now racing towards full matu- rity. Therefore, an understanding of this text will equip the reader with most of what they need to know in order to tackle any of the vast literature in this remarkable field, at least for a while yet.

Since we are aiming the discussion at active readers, we have matched every example with a corresponding snippet of (hyperlinked) Magma [BCP97] code 1 , where we take inspiration from the helpful Magma pairing tutorial by Dominguez Perez et al. [DKS09].

Early in the book we will develop examples that we then later extend with most of the things we learn in each chapter. This way we incrementally build a few real world snarks but over full fledged cryptographic systems that are nevertheless simple enough to be computed by pen and paper to illustrate all steps in grwat detail.

2.3 How to read this book

Books and papers to read: XXXXXXXXXXX

Software to try: XXXXXXXXXXXXXXXXXXXX

Correctly prescribing the best reading route for a beginner naturally requires individual diagnosis that depends on their prior knowledge and technical preparation.

2.4 Cryptological Systems

The science of information security is referred to as *cryptology*. In the broadest sense, it deals with encryption and decryption processes, with digital signatures, identification protocols, cryptographic hash functions, secrets sharing, electronic voting procedures and electronic money. EXPAND

2.5 SNARKS

2.6 complexity theory

Before we deal with the mathematics behind zero knowledge proof systems, we must first clarify what is meant by the runtime of an algorithm or the time complexity of an entire mathematical problem. This is particularly important for us when we analyze the various snark systems...

For the reader who is interested in complexity theory, we recommend, or example or , as well as the references contained therein.

2.6.1 Runtime complexity

The runtime complexity of an algorithm describes, roughly speaking, the amount of elementary computation steps that this algorithm requires in order to solve a problem, depending on the size of the input data.

Of course, the exact amount of arithmetic operations required depends on many factors such as the implementation, the operating system used, the CPU and many more. However, such accuracy is seldom required and is mostly meaningful to consider only the asymptotic computational effort.

In computer science, the runtime of an algorithm is therefore not specified in individual calculation steps, but instead looks for an upper limit which approximates the runtime as soon as the input quantity becomes very large. This can be done using the so-called *Landau notation* (also called big -$\mathcal{O}$-notation) A precise definition would, however, go beyond the scope of this work and we therefore refer the reader to .

For us, only a rough understanding of transit times is important in order to be able to talk about the security of crypographic systems. For example, $\mathcal{O}(n)$ means that the running time of the algorithm to be considered is linearly dependent on the size of the input set n, $\mathcal{O}(n^k)$ means that the running time is polynomial and $\mathcal{O}(2^n)$ stands for an exponential running time (chapter 2.4).

An algorithm which has a running time that is greater than a polynomial is often simply referred to as *slow*.

A generalization of the runtime complexity of an algorithm is the so-called *time complexity of a mathematical problem*, which is defined as the runtime of the fastest possible algorithm that can still solve this problem (chapter 3.1).

Since the time complexity of a mathematical problem is concerned with the runtime analysis of all possible (and thus possibly still undiscovered) algorithms, this is often a very difficult and deep-seated question .

For us, the time complexity of the so-called discrete logarithm problem will be important. This is a problem for which we only know slow algorithms on classical computers at the moment, but for which at the same time we cannot rule out that faster algorithms also exist.

STUFF ON CRYPTOGRAPHIC HASH FUNCTIOND

2.7 Software Used in This Book

2.7.1 Sagemath

It order to provide an interactive learning experience, and to allow getting hands-on with the concepts described in this book, we give examples for how to program them in the Sage programming language. Sage is a dialect of the learning-friendly programming language Python, which was extended and optimized for computing with, in and over algebraic objects. Therefore, we recommend installing Sage before diving into the following chapters.

The installation steps for various system configurations are described on the sage websit [1]. Note however that we use Sage version 9, so if you are using Linux and your package manager only contains version 8, you may need to choose a different installation path, such as using prebuilt binaries.

We recommend the interested reader, who is not familiar with sagemath to read on the many tutorial before starting this book. For example

[1] https://doc.sagemath.org/html/en/installation/index.html

Chapter 3

Arithmetics

3.1 Introduction

3.1.1 Aims and target audience

Sylvia: This subsection should be in the introduction of the book as it applies to the entire math part

Sylvia: This subsection should be in the introduction of the book as it applies to the entire math part

How much mathematics do you need to understand zero-knowledge proofs? The answer, of course, depends on the level of understanding you aim for. It is possible to describe zero-knowledge proofs without using mathematics at all; however, to read a foundational paper like **?** and to understand the internals of snark implementations, some knowledge of mathematics is needed to be able to follow the discussion.

zero-knowledge proofs

The goal of this chapter is therefore to enable a reader who is starting out with nothing more than basic high school algebra to be able to solve simple tasks in the mathematics underlying most zero knowledge proofs without the need of a computer.

Without a solid grounding in mathematics, someone who is interested in learning the concepts of zero-knowledge proofs, but who has never seen or dealt with, say, a prime field 4.3, or an elliptic curve 5, may quickly become overwhelmed. This is not so much due to the complexity of the mathematics needed, but rather because of the vast amount of technical jargon, unknown terms, and obscure symbols that quickly makes a text unreadable, even though the concepts themselves are not actually that complicted. As a result, the reader might either lose interest, or pick up some incoherent bits and pieces of knowledge that, in the worst case scenario, result in immature and unsecure code.

This is why we dedicated this chapter to explaining the mathematical foundations needed to understand the basic concepts underlying snark development. We encourage the reader who is not familiar with basic number theory and elliptic curves to take the time and read this and the following chapters, until they are able to solve at least a few exercises in each chapter.

If, on the other hand, you are already skilled in elliptic curve cryptography, feel free to skip this chapter and only come back to it for reference and comparison. Maybe the most interesting parts are XXX .

add references

We start our explanations at a very basic level, and only assume pre-existing knowledge of fundamental concepts like high school integer arithmetics. At the same time, we'll attempt to teach you to "think mathematically", and to show you that there are numbers and mathematical structures out there that appear to be very different from the things you learned about in high school, but on a deeper level, they are actually quite similar, as we will see in various examples below in this chapter.

To make it easier to memorize new concepts and symbols, we might frequently link to definitions in the beginning, but as to many links render a text unreadable, we will assume the reader will become familiar with definitions as the text proceeds at which point we will not link them anymore.

We want to stress, however that this introduction is informal, incomplete and optimized to enable the reader to understand zero-knowledge concepts as efficiently as possible. Our focus and design choices are to include as little theory as necessary, focusing on a wealth of numerical examples. We believe that such an informal, example-driven approach to learning mathematics may make it easier for beginners to digest the material in the initial stages.

For instance, as a beginner, you would probably find it more beneficial to first compute a simple toy **snark** with pen and paper all the way through, before actually developing real-world production-ready systems. In addition, it's useful to have a few simple examples in your head before getting started with reading actual academic papers.

However, in order to be able to derive these toy examples, some mathematical groundwork is needed. This chapter therefore will help the inexperienced reader to focus on what we believe is important, accompanied by exercises that you are encouraged to recompute yourself. Every section usually ends with a list of additional exercises in increasing order of difficulty, to help the reader memorize and apply the concepts.

3.1.2 The structure of this chapter

Sylvia: This subsection should be in the introduction of the book as it applies to the entire math part

Sylvia: This subsection should be in the introduction of the book as it applies to the entire math part

We start with a brief recapitulation of basic integer arithmetics like long division, the greatest common divisor and Euclid's algorithm. After that, we introduce modular arithmetics as **the most important skill** to compute our pen-and-paper examples. We then introduce polynomials, compute their analogs to integer arithmetics and introduce the important concept of Lagrange interpolation.

After this practical warm up, we introduce some basic algebraic terms like groups and fields, because those terms are used very frequently in academic papers relating to zero-knowledge proofs. The beginner is advised to memorize those terms and think about them: Why are they defined the way they are? Can you come up with a better definition? How and why is every rule important and what happens if that rule is dropped?

We define these terms in the general abstract way of mathematics, hoping that the non mathematical trained reader will gradually learn to become comfortable with this style. We then give basic examples and do basic computations with these examples to get familiar with the concepts.

3.2 Integer Arithmetics

In a sense, integer arithmetics is at the heart of large parts of modern cryptography. Fortunately, most readers will probably remember integer arithmetics from school. It is, however, important that you can confidently apply those concepts to understand and execute computations in the many pen-and-paper examples that form an integral part of the MoonMath Manual. We will therefore recapitulate basic arithmetics concepts to refresh your memory and fill any knowledge gaps.

In what follows, we use many mathematical notations, which we summerized in the following table 3.2:

Notatio

Symbol	Meaning of Symbol
$=$	equals
$:=$	defining the symbol on the r
$\in$	element from a set
$\Leftrightarrow$	logical equivalence
$\sum_{j=n}^{k} a_j$	summation

Sylvia: We should put this table into the same introduction as 3.1.1

Sylvia: We should put this table into the same introduction as 3.1.1

In this book, we use the symbol $\mathbb{Z}$ as a short description for the set of all **integers**, that is we write:

$$\mathbb{Z} := \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} \tag{3.1}$$

Integers are also known as **whole numbers**, that is, numbers that can be written without fractional parts. Examples of numbers that are **not** integers are $\frac{2}{3}$, 1.2 and -1280.006.

If $a \in \mathbb{Z}$ is an integer, then we write $|a|$ for the **absolute value** of a, that is, the the non-negative value of a without regard to its sign:

$$|4| = 4 \tag{3.2}$$
$$|-4| = 4 \tag{3.3}$$

We use the symbol $\mathbb{N}$ for the set of all positive integers, usually called the set of **natural numbers** and $\mathbb{N}_0$ for the set of all non negative integers. So whenever you see the symbol $\mathbb{N}$, think of the set of all positive integers excluding the number 0:

$$\mathbb{N} := \{1, 2, 3, \ldots\} \qquad \mathbb{N}_0 := \{0, 1, 2, 3, \ldots\}$$

In addition, we use the symbol $\mathbb{Q}$ for the set of all **rational numbers**, which can be represented as the set of all fractions $\frac{n}{m}$, where $n \in \mathbb{Z}$ is an integer and $m \in \mathbb{N}$ is a natural number, such that there is no other fraction $\frac{n'}{m'}$ and natural number $k \in \mathbb{N}$ with $k \neq 1$ and

@jan @anna double check this definition. Is it clear enough? Proper definition requires the concept of equivalance or coprimeness first

$$\frac{n}{m} = \frac{k \cdot n'}{k \cdot m'} \tag{3.4}$$

The sets $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$ have a notion of addition and multiplication defined on them. Most of us are probably able to do many integer computations in our head, but this gets more and more difficult as these increase in complexity. We will frequently invoke the SageMath system (2.7.1) for more complicated computations (We define rings and fields later in this book):

```
sage: ZZ # A sage notation for the integers
Integer Ring
sage: NN # A sage notation for the natural numbers
Non negative integer semiring
sage: QQ # A sage notation for the rational numbers
Rational Field
sage: ZZ(5) # Get an element from the integers
5
```

```
sage: ZZ(5) + ZZ(3)
8
sage: ZZ(5) * NN(3)
15
sage: ZZ.random_element(10**50)
14015294682049863751664889394150639107472169893405
sage: ZZ(27713).str(2) # Binary string representation
110110001000001
sage: NN(27713).str(2) # Binary string representation
110110001000001
sage: ZZ(27713).str(16) # Hexadecimal string representation
6c41
```

One set of numbers that is of particular interest to us is the set of **prime numbers**, which are natural numbers $p \in \mathbb{N}$ with $p \geq 2$, which are only divisible by themself and by 1. All prime numbers apart from the number 2 are called **odd** prime numbers. We write $\mathbb{P}$ for the set of all prime numbers and $\mathbb{P}_{\geq 3}$ for the set of all odd prime numbers. The set of prime numbers $\mathbb{P}$ is an infinite set and can be ordered according to size, which means that for any prime number $p \in \mathbb{P}$ one can always find another prime number $p' \in \mathbb{P}$ with $p < p'$. It follows thar there is no largest prime number. Since prime numbers can be ordered by size, we can write them as follows:

$$2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,\ldots \tag{3.5}$$

As the **fundamental theorem of arithmetics** tells us, prime numbers are, in a certain sense, the basic building blocks from which all other natural numbers are composed. To see that, let $n \in \mathbb{N}$ be any natural number with $n > 1$. Then there are always prime numbers $p_1, p_2, \ldots, p_k \in \mathbb{P}$, such that

$$n = p_1 \cdot p_2 \cdot \ldots \cdot p_k \,. \tag{3.6}$$

This representation is unique for each natural number (except for the order of the factors) and is called the **prime factorization** of n.

Example 1 (Prime Factorization). To see what we mean by prime factorization of a number, let's look at the number $504 \in \mathbb{N}$. To get its prime factors, we can successively divide it by all prime numbers in ascending order starting with 2:

$$504 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 7$$

We can double check our findings invoking Sage, which provides an algorithm to factor natural numbers:

```
sage: n = NN(504)
sage: factor(n)
2^3 * 3^2 * 7
```

The computation from the previous example reveals an important observation: Computing the factorization of an integer is computationally expensive, because we have to divide repeadly by all prime numbers smaller then the number itself until all factors are prime numbers themself. From this, an important question arises: How fast can we compute the prime factorization of a natural number? This question is the famous **integer factorization problem** and, as far as we know, there is currently no method known that can factor integers much faster then the naive approach that just divides the given number by all prime numbers in ascending order.

On the other hand computing the product of a given set of prime numbers, is fast (just multiply all factors) and this simple observation implies that the two processes "prime number multiplication" on the one side and its inverse process "natural number factorization" have very different computational costs. The factorization problem is therefore an example of a so-called **one-way function**: An invertible function that is easy to compute in one direction, but hard to compute in the other direction.

It should be pointed out, however, that the American mathematician Peter W. Shor developed an algorithm in **?** which can calculate the prime factor ization of a natural number in polynomial time on a quantum computer. The consequence of this is that cryptosystems, which are based on the prime factor problem, are unsafe as soon as practically usable quantum computers become available .

Do we even need this quantum computing excurse?

Exercise 1. What is the absolute value of the integers -123, 27 and 0?

Exercise 2. Compute the factorization of 30030 and double check your results using Sage.

Exercise 3. Consider the following equation $4 \cdot x + 21 = 5$. Compute the set of all solutions for x under the following alternative assumptions:

1. The equation is defined over the natural numbers.
2. The equation is defined over the integers.

Exercise 4. Consider the following equation $2x^3 - x^2 - 2x = -1$. Compute the set of all solutions x under the following assumptions:

1. The equation is defined over the natural numbers.
2. The equation is defined over the integers.
3. The equation is defined over the rational numbers.

Euclidean Division As we know from high school mathematics, integers can be added, subtracted and multiplied and the result is guranteed to always be an integer again. On the contrary division in the commonly understood sense is not defined for integers, as, for example, 7 divided by 3 will not be an integer again. However it is always possible to divide any two integers if we consider division with remainder. So for example 7 divided by 3 is equal to 2 with a remainder of 1, since $7 = 2 \cdot 3 + 1$.

It is the content of this section to introduced division with remainder for integers which is usually called **Euclidean division**. It is an essential technique underlying many concepts in this book. The precise definition is as follows:

Let $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ be two integers with $b \neq 0$. Then there is always another integer $m \in \mathbb{Z}$ and a natural number $r \in \mathbb{N}$, with $0 \leq r < |b|$ such that

$$a = m \cdot b + r \tag{3.7}$$

This decomposition of a given b is called **Euclidean division**, where a is called the **dividend**, b is called the **divisor**, m is called the **quotient** and r is called the **remainder**. It can be shown that both the quotient and the remainder always exist and are unique, as long as the divisor is different from 0.

@jan. You wrote: a and b are required to be non-zero in the def-

Notation and Symbols 1. Suppose that the numbers a, b, m and r satisfy equation (3.7). Then we often write

$$a \text{ div } b := m, \qquad a \text{ mod } b := r \tag{3.8}$$

to describe the quotient and the remainder of the Euclidean division. We also say that an integer a is divisible by another integer b if $a \text{ mod } b = 0$ holds. In this case we also write $b|a$.

So, in a nutshell Euclidean division is a process of dividing one integer by another in a way that produces a quotient and a non-negative remainder, the latter of which is smaller than the absolute value of the divisor.

A special situation occurs whenever the remainder is zero, because in this case the dividend is divisible by the divisor. Our notation $b|a$ reflects that.

Example 2. Applying Euclidean division and our previously defined notation 3.8 to the dividend -17 and the divisor 4, we get

$$-17 \text{ div } 4 = -5, \qquad -17 \text{ mod } 4 = 3$$

because $-17 = -5 \cdot 4 + 3$ is the Euclidean division of -17 and 4 (the remainder is, by definition, a non-negative number). In this case 4 does not divide -17, as the reminder is not zero. The truth value of the expression $4|-17$ therefore is FALSE. On the other hand, the truth value of $4|12$ is TRUE, since 4 divides 12, as $12 \text{ mod } 4 = 0$. We can invoke sage to do the computation for us. We get the following:

```
sage: ZZ(-17) // ZZ(4) # Integer quotient
-5
sage: ZZ(-17) % ZZ(4) # remainder
3
sage: ZZ(4).divides(ZZ(-17)) # self divides other
False
sage: ZZ(4).divides(ZZ(12))
True
```

You wrote: if these should only satisfy the equation, why use definition symbols (:=) and not equality symbols (=)? But this is a definition the symbol a div b IS DEFINED to be the number b...

Remark 1. In 3.8 we defined the notation of a div b and a mod b, in terms of Euclidean division. It should be noted however that many programing languages like Phyton and Sage, implement both the operator $(/)$ as well as the operator $(\%)$ differently. Programers should be aware of this, as the discrepancy between the mathematical notation and the implementation in programing languages might become the source of subtle bugs in implementations of cryptographic primitives.

To give an example consider the the dividend -17 and the divisor -4. Note that in contrast to the previous example 2, we have a negative divisor. According to our definition we have

$$-17 \text{ div } -4 = 5, \qquad -17 \text{ mod } -4 = 3$$

because $-17 = 5 \cdot (-4) + 3$ is the Euclidean division of -17 and -4 (the remainder is, by definition, a non-negative number). However using the operators $(/)$ and $(\%)$ in Sage we get

```
sage: ZZ(-17) // ZZ(-4) # Integer quotient
4
sage: ZZ(-17) % ZZ(-4) # remainder
-1
```

Methods to compute Euclidean division for integers are called **integer division algorithms**. Probably the best known algorithm is the so-called **long division**, which most of us might have learned in school.

As long division is the standard method used for pen-and-paper division of multi-digit numbers expressed in decimal notation, the reader should become familiar with it as we use it throughout this book when we do simple pen-and-paper computations. However, instead of defining the algorithm formally, we rather give some examples that will hopefully make the process clear.

In a nutshell, the algorithm loops through the digits of the dividend from the left to right, subtracting the largest possible multiple of the divisor (at the digit level) at each stage; the multiples then become the digits of the quotient, and the remainder is the first digit of the dividend.

Example 3 (Integer Long Division). To give an example of integer long division algorithm, lets divide the integer $a = 143785$ by the number $b = 17$. Our goal is therefore to find solutions to equation 3.7, that is, we need to find the quotient $m \in \mathbb{Z}$ and the remainder $r \in \mathbb{N}$ such that $143785 = m \cdot 17 + r$. Using a notation that is mostly used in Commonwealth countries, we compute as follows:

$$\begin{array}{r@{\,}r}
 & 8457 \\
17 \,\big) & \overline{143785} \\
 & \underline{136} \\
 & 77 \\
 & \underline{68} \\
 & 98 \\
 & \underline{85} \\
 & 135 \\
 & \underline{119} \\
 & 16
\end{array} \tag{3.9}$$

We therefore get $m = 8457$ as well as $r = 16$ and indeed we have $143785 = 8457 \cdot 17 + 16$, which we can double check invoking Sage:

```
sage: ZZ(143785).quo_rem(ZZ(17)) # Euclidean Division
(8457, 16)
sage: ZZ(143785) == ZZ(8457)*ZZ(17) + ZZ(16) # check
True
```

Exercise 5 (Integer Long Division). Find an $m \in \mathbb{Z}$ as well as an $r \in \mathbb{N}$ such that $a = m \cdot b + r$ holds for the following pairs $(a,b) = (27,5)$, $(a,b) = (27,-5)$, $(a,b) = (127,0)$, $(a,b) = (-1687,11)$ and . In which cases are your solutions unique?

$(a,b) = (0,7)$

Exercise 6 (Long Division Algorithm). Write an algorithm that computes integer long division and handling all edge cases properly.

The Extended Euclidean Algorithm One of the most critical parts in this book is modular arithmetics 3.3 and its application in the computations in so-called **prime fields**, as we explain in 4.3. In modular arithmetics, it is sometimes possible to define division and multiplicative inverses of numbers that is very different from inverses as we know them from other systems like rational numbers.

modular arithmetics

division

multiplicative inverses

However, to actually compute those inverses, we have to get familiar with the so-called **extended Euclidean algorithm**. A few more terms are necessary to explain the concept: The **greatest common divisor** (GCD) of two nonzero integers a and b is the greatest non-zero natural number d such that d divides both a and b, that is, $d|a$ as well as $d|b$. We write $gcd(a,b) := d$ for this number. In addition, two natural numbers are called **relative primes** or **coprimes**, if their greatest common divisor is 1 .

@jan. You wrote "For two things to be co-prime means that they don't share any prime factors. While this is equivalent to a gcd of 1, defining it that way gets it backwards; it's not "two things are comprime if they have a gcd of 1, and hence coprime means that they have common prime factors", but the other way around." Both properties are equivalent and hence both can be the definition. There is no preference really.

The Extended Euclidean algorithm is a method to calculate the greatest common divisor of two natural numbers a and $b \in \mathbb{N}$, as well as two additional integers $s,t \in \mathbb{Z}$, such that the following equation holds:

$$gcd(a,b) = s \cdot a + t \cdot b \tag{3.10}$$

The following pseudocode shows in detail how to calculate these numbers with the extended Euclidean algorithm:

Algorithm 1 Extended Euclidean Algorithm

Require: $a,b \in \mathbb{N}$ with $a \geq b$
  procedure EXT-EUCLID(a,b)
    $r_0 \leftarrow a$
    $r_1 \leftarrow b$
    $s_0 \leftarrow 1$
    $s_1 \leftarrow 0$
    $k \leftarrow 1$
    while $r_k \neq 0$ **do**
      $q_k \leftarrow r_{k-1}$ div r_k
      $r_{k+1} \leftarrow r_{k-1} - q_k \cdot r_k$
      $s_{k+1} \leftarrow s_{k-1} - q_k \cdot s_k$
      $k \leftarrow k+1$
    end while
    return $gcd(a,b) \leftarrow r_{k-1}$, $s \leftarrow s_{k-1}$ and $t := (r_{k-1} - s_{k-1} \cdot a)$ div b
  end procedure
Ensure: $gcd(a,b) = s \cdot a + t \cdot b$

The algorithm is simple enough to be done effectively in pen-and-paper examples, where it is common to write it as a table where the rows represent the while-loop and the columns represent the values of the the array r, s and t with index k. The following example provides a simple execution:

Example 4. To illustrate the algorithm, let's apply it to the numbers $a = 12$ and $b = 5$. Since $12, 5 \in \mathbb{N}$ as well as $12 \geq 5$ all requirements are met and we compute as follows:

k	r_k	s_k	$t_k = (r_k - s_k \cdot a)$ div b
0	12	1	0
1	5	0	1
2	2	1	-2
3	1	-2	5

From this we can see that 12 and 5 are relatively prime (coprime), since their greatest common divisor is $gcd(12,5) = 1$ and that the equation $1 = (-2) \cdot 12 + 5 \cdot 5$ holds. We can also invoke sage to double check our findings:

```
sage: ZZ(12).xgcd(ZZ(5)) # (gcd(a,b),s,t)
```

40

```
(1, -2, 5)
```

41

Exercise 7 (Extended Euclidean Algorithm). Find integers $s,t \in \mathbb{Z}$ such that $gcd(a,b) = s\cdot a + t\cdot b$ holds for the following pairs $(a,b) = (45,10)$, $(a,b) = (13,11)$, $(a,b) = (13,12)$. What pairs (a,b) are coprime?

Exercise 8 (Towards Prime fields). Let $n \in \mathbb{N}$ be a natural number and p a prime number, such that $n < p$. What is the greatest common divisor $gcd(p,n)$?

Exercise 9. Find all numbers $k \in \mathbb{N}$ with $0 \leq k \leq 100$ such that $gcd(100,k) = 5$.

Exercise 10. Show that $gcd(n,m) = gcd(n+m,m)$ for all $n,m \in \mathbb{N}$.

3.3 Modular arithmetic

In mathematics, **modular arithmetic** is a system of arithmetic for integers, where numbers "wrap around" when reaching a certain value, much like calculations on a clock wrap around whenever the value exceeds the number 12. For example, if the clock shows that it is 11 o'clock, then 20 hours later it will be 7 o'clock, not 31 o'clock. The number 31 has no meaning on a normal clock that shows hours.

The number at which the wrap occurs is called the **modulus**. Modular arithmetics generalizes the clock example to arbitrary moduli and studies equations and phenomena that arise in this new kind of arithmetics. It is of central importance for understanding most modern crypto systems, in large parts because the exponentiation function has an inverse with respect to certain moduli that is hard to compute . In addition, we will see that it provides the foundation of what is called finite fields ().

exponentiation function

@jan you wrote: "This is backwards. The DLP is the problem of computing a logarithm in finite fields, so first and foremost you can use mod to get finite fields, and within that system, computing the inverse to an exponentiation is hard. I think it may make sense to

Although modular arithmetic appears very different from ordinary integer arithmetic that we are all familiar with, we encourage the interested reader to work through the example and to discover that, once they get used to the idea that this is a new kind of calculations, it will seem much less daunting.

Congruence In what follows, let $n \in \mathbb{N}$ with $n \geq 2$ be a fixed natural number that we will call the **modulus** of our modular arithmetics system. With such an n given, we can then group integers into classes, by saying that two integers are in the same class, whenever their Euclidean division 3.2 by n will give the same remainder. We then say that two numbers are **congruent** whenever they are in the same class.

Example 5. If we choose $n = 12$ as in our clock example, then the integers -7, 5, 17 and 29 are all congruent with respect to 12, since all of them have the remainder 5 if we perform Euclidean division on them by 12. In the picture of an analog 12-hour clock, starting at 5 o'clock, when we add 12 hours we are again at 5 o'clock, representing the number 17. On the other hand, when we subtract 12 hours, we are at 5 o'clock again, representing the number -7.

We can formalize this intuition of what congruence should be into a proper definition utilizing Euclidean division (as explained previously in 3.2): Let $a, b \in \mathbb{Z}$ be two integers and $n \in \mathbb{N}$ a natural number. Then a and b are said to be **congruent with respect to the modulus** n, if and only if the following equation holds

$$a \bmod n = b \bmod n \tag{3.11}$$

If, on the other hand, two numbers are not congruent with respect to a given modulus n, we call them **incongruent** w.r.t. n.

A **congruence** is then nothing but an equation "up to congruence", which means that the equation only needs to hold if we take the modulus on both sides. In which case we write

$$a \equiv b \pmod{n} \tag{3.12}$$

Exercise 11. Which of the following pairs of numbers are congruent with respect to the modulus 13: $(5,19)$, $(13,0)$, $(-4,9)$, $(0,0)$.

Exercise 12. Find all integers x, such that the congruence $x \equiv 4 \pmod{6}$ is satisfied.

Modular Arithmetics One particulary useful thing about congruencies is that we can do calculations (arithmetics), much like we can with integer equations. That is, we can add or multiply numbers on both sides. The main difference is probably that the congruence $a \equiv b \pmod{n}$ is only equivalent to the congruence $k \cdot a \equiv k \cdot b \pmod{n}$ for some non-zero integer $k \in \mathbb{Z}$, whenever k and the modulus n are coprime. The following list gives a set of useful rules:

Suppose that integers $a_1, a_2, b_1, b_2, k \in \mathbb{Z}$ are given Then the following arithmetic rules hold for congruencies:

- $a_1 \equiv b_1 \pmod{n} \Leftrightarrow a_1 + k \equiv b_1 + k \pmod{n}$ (compatibility with translation)
- $a_1 \equiv b_1 \pmod{n} \Rightarrow k \cdot a_1 \equiv k \cdot b_1 \pmod{n}$ (compatibility with scaling)
- $ggt(k,n) = 1$ and $k \cdot a_1 \equiv k \cdot b_1 \pmod{n} \Rightarrow a_1 \equiv b_1 \pmod{n}$
- $k \cdot a_1 \equiv k \cdot b_1 \pmod{k \cdot n} \Rightarrow a_1 \equiv b_1 \pmod{n}$
- $a_1 \equiv b_1 \pmod{n}$ and $a_2 \equiv b_2 \pmod{n} \Rightarrow a_1 + a_2 \equiv b_1 + b_2 \pmod{n}$ (compatibility with addition)
- $a_1 \equiv b_1 \pmod{n}$ and $a_2 \equiv b_2 \pmod{n} \Rightarrow a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{n}$ (compatibility with multiplication)

Other rules, such as compatibility with subtraction and exponentiation, follow from the rules above. For example, compatibility with subtraction follows from compatibility with scaling by $k = -1$ and compatibility with addition.

Another property of congruencies, not known in the traditional arithmetics of integers is the **Fermat's Little Theorem**. In simple words, it states that, in modular arithmetics, every number raised to the power of a prime number modulus is congruent to the number itself. Or, to be more precise, if $p \in \mathbb{P}$ is a prime number and $k \in \mathbb{Z}$ is an integer, then:

$$k^p \equiv k \pmod{p}, \tag{3.13}$$

If k is coprime to p, then we can divide both sides of this congruence by k and rewrite the expression into the equivalent form

$$k^{p-1} \equiv 1 \pmod{p} \tag{3.14}$$

The following sage code computes example effects of Fermat's little theorem and highlights the effects of the exponent k being coprime and not coprime to p:

```
sage: ZZ(137).gcd(ZZ(64))
1
```

```
sage: ZZ(64)** ZZ(137) % ZZ(137) == ZZ(64) % ZZ(137)
True
sage: ZZ(64)** ZZ(137-1) % ZZ(137) == ZZ(1) % ZZ(137)
True
sage: ZZ(1918).gcd(ZZ(137))
137
sage: ZZ(1918)** ZZ(137) % ZZ(137) == ZZ(1918) % ZZ(137)
True
sage: ZZ(1918)** ZZ(137-1) % ZZ(137) == ZZ(1) % ZZ(137)
False
```

Now, since for the sake of readers who have never encountered modular arithmetics before, let's compute an example that contains most of the concepts described in this section:

Example 6. Assume that we consider the modulus 6 and that our task is to solve the following congruence for $x \in \mathbb{Z}$

I dissagre on Jan's comment here. Its important for a newbe to understand computation is similar to "normal" numbers

$$7 \cdot (2x + 21) + 11 \equiv x - 102 \quad (\bmod 6)$$

As many rules for congruencies are more or less same as for integers, we can proceed in a similar way as we would if we had an equation to solve. Since both sides of a congruence contain ordinary integers, we can rewrite the left side as follows: $7 \cdot (2x+21)+11 = 14x+147+11 = 14x+158$. We can therefore rewrite the congruence into the equivalent form

$$14x + 158 \equiv x - 102 \quad (\bmod 6)$$

In the next step we want to shift all instances of x to left and every other term to the right. So we apply the"compatibility with translation" rules two times. In a first step we choose $k = -x$ and in a second step we choose $k = -158$. Since "compatibility with translation" transforms a congruence into an equivalent form, the solution set will not change and we get

$$\begin{aligned}
14x + 158 \equiv x - 102 \quad (\bmod 6) &\Leftrightarrow \\
14x - x + 158 - 158 \equiv x - x - 102 - 158 \quad (\bmod 6) &\Leftrightarrow \\
13x \equiv -260 \quad (\bmod 6) &
\end{aligned}$$

If our congruence would just be a normal integer equation, we would divide both sides by 13 to get $x = -20$ as our solution. However, in case of a congruence, we need to make sure that the modulus and the number we want to divide by are coprime first – only then will we get an equivalent expression (See rule XXX). So we need to find the greatest common divisor $gcd(13,6)$. Since 13 is prime and 6 is not a multiple of 13, we know that$gcd(13,6) = 1$, so these numbers are indeed coprime. We therefore compute

$$13x \equiv -260 \quad (\bmod 6) \Leftrightarrow x \equiv -20 \quad (\bmod 6)$$

Our task is now to find all integers x, such that x is congruent to -20 with respect to the modulus 6. So we have to find all x such

$$x \bmod 6 = -20 \bmod 6$$

Since $-4 \cdot 6 + 4 = -20$ we know $-20 \bmod 6 = 4$ and hence we know that $x = 4$ is a solution to this congruence. However, 22 is another solution since $22 \bmod 6 = 4$ as well, and so is -20. In fact, there are infinitely many solutions given by the set

$$\{\ldots, -8, -2, 4, 10, 16, \ldots\} = \{4 + k \cdot 6 \mid k \in \mathbb{Z}\}$$

Putting all this together, we have shown that the every x from the set $\{x = 4 + k \cdot 6 \mid k \in \mathbb{Z}\}$ is a solution to the congruence $7 \cdot (2x + 21) + 11 \equiv x - 102 \pmod{6}$. We double ckeck for, say, $x = 4$ as well as $x = 4 + 12 \cdot 6 = 76$ using sage:

```
sage: (ZZ(7)* (ZZ(2)*ZZ(4) + ZZ(21)) + ZZ(11))  % ZZ(6) == (ZZ(4) - ZZ(102))  % ZZ(6)
True
sage: (ZZ(7)* (ZZ(2)*ZZ(76) + ZZ(21)) + ZZ(11))  % ZZ(6) == (ZZ(76) - ZZ(102))  % ZZ(6)
True
```

Readers who had not been familiar with modular arithmetics until now and who might be discouraged by how complicated modular arithmetics seems at this point, should keep two things in mind. First, computing congruencies in modular arithmetics is not really more complicated than computations in more familiar number systems (e.g. fractional numbers), it is just a matter of getting used to it. Second, the theory of prime fields4.3 takes a different view on modular arithmetics with the attempt to simplify matters. In other words, once we understand prime field arithmetics, things become conceptually cleaner and more easy to compute.

Exercise 13. Consider the modulus 13 and find all solutions $x \in \mathbb{Z}$ to the following congruence $5x + 4 \equiv 28 + 2x \pmod{13}$

Exercise 14. Consider the modulus 23 and find all solutions $x \in \mathbb{Z}$ to the following congruence $69x \equiv 5 \pmod{23}$

Exercise 15. Consider the modulus 23 and find all solutions $x \in \mathbb{Z}$ to the following congruence $69x \equiv 46 \pmod{23}$

Exercise 16. Let a, b, k be integers, such that $a \equiv b \pmod{n}$ holds. Show $a^k \equiv b^k \pmod{n}$.

Exercise 17. Let a, n be integers, such that a and n are not coprime. For which $b \in \mathbb{Z}$ does the congruence $a \cdot x \equiv b \pmod{n}$ have a solution x and how does the solution set look in that case?

The Chinese Remainder Theorem We have seen how to solve congruencies in modular arithmetic. However, one question that remains is how to solve systems of congruencies with different moduli? The answer is given by the **Chinese reimainder theorem**, which states that for any $k \in \mathbb{N}$ and coprime natural numbers $n_1, \ldots n_k \in \mathbb{N}$ as well as integers $a_1, \ldots a_k \in \mathbb{Z}$, the so-called **simultaneous congruences**

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\cdots \\ x &\equiv a_k \pmod{n_k} \end{aligned} \tag{3.15}$$

has a solution, and all possible solutions of this congruence system are congruent modulo the product $N = n_1 \cdot \ldots \cdot n_k$.[1] In fact, the following algorithm computes the solution set:

check algorithm floating

[1]This is the classical Chinese remainder theorem as it was already known in ancient China. Under certain circumstances, the theorem can be extended to non-coprime moduli $n_1, \ldots, n_k$ but this is beyond the scope of this book. Interested readers should consult XXX add references

Algorithm 2 Chinese Remainder Theorem

Require: $, k \in \mathbb{Z}$, $j \in \mathbb{N}_0$ and $n_0, \ldots, n_{k-1} \in \mathbb{N}$ coprime
  procedure CONGRUENCE-SYSTEMS-SOLVER($a_0, \ldots, a_{k-1}$)
    $N \leftarrow n_0 \cdot \ldots \cdot n_{k-1}$
    while $j < k$ **do**
      $N_j \leftarrow N/n_j$
      $(\_, s_j, t_j) \leftarrow EXT-EUCLID(N_j, n_j)$ $\triangleright\ 1 = s_j \cdot N_j + t_j \cdot n_j$
    end while
    $x' \leftarrow \sum_{j=0}^{k-1} a_j \cdot s_j \cdot N_j$
    $x \leftarrow x' \bmod N$
    return $\{x + m \cdot N \mid m \in \mathbb{Z}\}$
  end procedure
Ensure: $\{x + m \cdot N \mid m \in \mathbb{Z}\}$ is the complete solution set to 3.15.

Example 7. To illustrate how to solve simultaneous congruences using the Chinese remainder theorem, let's look at the following system of congruencies:

$$\begin{aligned} x &\equiv 4 \quad (\bmod 7) \\ x &\equiv 1 \quad (\bmod 3) \\ x &\equiv 3 \quad (\bmod 5) \\ x &\equiv 0 \quad (\bmod 11) \end{aligned}$$

Clearly all moduli are coprime and we have $N = 7 \cdot 3 \cdot 5 \cdot 11 = 1155$, as well as $N_1 = 165$, $N_2 = 385$, $N_3 = 231$ and $N_4 = 105$. From this we calculate with the extended Euclidean algorithm

$$\begin{array}{rrcr} 1 = & 2 \cdot 165 & + & -47 \cdot 7 \\ 1 = & 1 \cdot 385 & + & -128 \cdot 3 \\ 1 = & 1 \cdot 231 & + & -46 \cdot 5 \\ 1 = & 2 \cdot 105 & + & -19 \cdot 11 \end{array}$$

so we have $x = 4 \cdot 2 \cdot 165 + 1 \cdot 1 \cdot 385 + 3 \cdot 1 \cdot 231 + 0 \cdot 2 \cdot 105 = 2398$ as one solution. Because $2398 \bmod 1155 = 88$ the set of all solutions is $\{\ldots, -2222, -1067, 88, 1243, 2398, \ldots\}$. We can invoke Sage's computation of the Chinese Remainder Theorem (CRT) to double check our findings:

```
sage: CRT_list([4,1,3,0], [7,3,5,11])
88
```

Remainder Classes As we have seen in various examples before, computing congruencies can be cumbersome and solution sets are large in general. It is therefore advantaegous to find some kind of simplification for modular arithmetic.

Fortunately, this is possible and relatively straightforward once we consider all integers that have the same remainder with respect to a given modulus n in Euclidean division to be equivalent. With such a definition of equivalence in mind we then identify each set of numbers with equal remainder with that remainder and call it a **remainder class** or **residue class** in modulo n arithmetics .

@jan: does this satisfies your comment?

It then follows from the properties of Euclidean division that there are exactly n different remainder classes for every modulus n and that integer addition and multiplication can be projected to a new kind of addition and multiplication on those classes.

Roughly speaking, the new rules for addition and multiplication are then computed by taking any element of the first equivalence class and some element of the second, then add or multiply them in the usual way and see which equivalence class the result is contained in. The following example makes this abstract description more concrete:

Example 8 (Arithmetics modulo 6). Choosing the modulus $n = 6$, we have six equivalence classes of integers which are congruent modulo 6 (they have the same remainder when divided by 6) and when we identify each of those remainder classes with the remainder, we get the following identification:

$$\begin{aligned}
0 &:= \{\ldots,-6,0,6,12,\ldots\}\\
1 &:= \{\ldots,-5,1,7,13,\ldots\}\\
2 &:= \{\ldots,-4,2,8,14,\ldots\}\\
3 &:= \{\ldots,-3,3,9,15,\ldots\}\\
4 &:= \{\ldots,-2,4,10,16,\ldots\}\\
5 &:= \{\ldots,-1,5,11,17,\ldots\}
\end{aligned}$$

Now to compute the addition of those equivalence classes, say $2+5$, one chooses arbitrary elements from both sets, say 14 and -1, adds those numbers in the usual way and then looks at the equivalence class of the result.

So we get $14+(-1) = 13$, and 13 is in the equivalence class (of) 1. Hence we find that $2+5 = 1$ in modular 6 arithmetics, which is a more readable way to write the congruence $2+5 \equiv 1 \pmod 6$.

Applying the same reasoning to all equivalence classes, addition and multiplication can be transferred to equivalence classes. The results for modulus 6 arithmetics are summarized in the following addition and multiplication tables:

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

·	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

This way, we have defined a new arithmetic system that contains just 6 numbers and comes with its own definition of addition and multiplication. It is called**modular 6 arithmetics** and written as $\mathbb{Z}_6$.

To see why such an identification of a congruence class with its remainder is useful and actually simplifies congruence computations a lot, lets go back to the congruence from example 6 again:

$$7\cdot(2x+21)+11 \equiv x-102 \pmod 6 \tag{3.16}$$

As shown in example 6, the arithmetics of congruencies can deviate from ordinary arithmetics: For example, division needs to check whether the modulus and the dividend are co-primes, and solutions are not unique in general.

We can rewrite this congruence as an **equation** over our new arithmetic type $\mathbb{Z}_6$ by **projecting onto the remainder classes**. In particular, since $7 \bmod 6 = 1$, $21 \bmod 6 = 3$, $11 \bmod 6 = 5$ and $102 \bmod 6 = 0$ we have

$$\begin{aligned}
&7\cdot(2x+21)+11 \equiv x-102 \pmod 6 \text{ over } \mathbb{Z}\\
&\qquad\qquad\qquad\qquad \Leftrightarrow\ 1\cdot(2x+3)+5 = x \text{ over } \mathbb{Z}_6
\end{aligned}$$

We can use the multiplication and addition table above to solves the equation on the right like we would solve normal integer equations:

$$\begin{aligned}
1\cdot(2x+3)+5 &= x & & \\
2x+3+5 &= x & & \text{\# addition-table: } 3+5=2\\
2x+2 &= x & & \text{\# add 4 and } -x \text{ on both sides}\\
2x+2+4-x &= x+4-x & & \text{\# addition-table: } 2+4=0\\
x &= 4 & &
\end{aligned}$$

As we can see, despite the somewhat unfamiliar rules of addition and multiplication, solving congruencies this way is very similar to solving normal equations. And, indeed, the solution set is identical to the solution set of the original congruence, since 4 is identified with the set $\{4+6\cdot k \mid k\in\mathbb{Z}\}$.

We can invoke Sage to do computations in our modular 6 arithmetics type. This is particularly useful to double-check our computations:

```
sage: Z6 = Integers(6)
sage: Z6(2) + Z6(5)
1
sage: Z6(7)*(Z6(2)*Z6(4)+Z6(21))+Z6(11) == Z6(4) - Z6(102)
True
```

Jargon 1 (k-bit modulus). In cryptographic papers, we sometimes read phrases like"[...] using a 4096-bit modulus". This means that the underlying modulus n of the modular arithmetic used in the system has a binary representation with a length of 4096 bits. In contrast, the number 6 has the binary representation 110 and hence our example 8 describes a 3-bit modulus arithmetics system.

Exercise 18. Define $\mathbb{Z}_{13}$ as the the arithmetics modulo 13 analog to example 8. Then consider the congruence from exercise 13 and rewrite it into an equation in $\mathbb{Z}_{13}$

Modular Inverses As we know, integers can be added, subtracted and multiplied so that the result is also an integer, but this is not true for the division of integers in general: for example, $3/2$ is not an integer anymore. To see why this is, from a more theoretical perspective, let us consider the definition of a multiplicative inverse first. When we have a set that has some kind of multiplication defined on it and we have a distinguished element of that set that behaves neutrally with respect to that multiplication (doesn't change anything when multiplied with any other element), then we can define **multiplicative inverses** in the following way:

Let S be our set that has some notion $a\cdot b$ of multiplication and a **neutral element** $1\in S$, such that $1\cdot a=a$ for all elements $a\in S$. Then a **multiplicative inverse** a^{-1} of an element $a\in S$ is defined as follows:

$$a\cdot a^{-1}=1 \tag{3.17}$$

Informally speaking, the definition of a multiplicative inverse is means that it "cancels" the original element to give 1 when they are multiplied.

Numbers that have multiplicative inverses are of particular interest, because they immediately lead to the definition of division by those numbers. In fact, if a is number such that the multiplicative inverse a^{-1} exists, then we define **division** by a simply as multiplication by the inverse:

$$\frac{b}{a} := b \cdot a^{-1} \tag{3.18}$$

Example 9. Consider the set of rational numbers, also known as fractions, $\mathbb{Q}$. For this set, the neutral element of multiplication is 1, since $1 \cdot a = a$ for all rational numbers. For example, $1 \cdot 4 = 4$, $1 \cdot \frac{1}{4} = \frac{1}{4}$, or $1 \cdot 0 = 0$ and so on.

Every rational number $a \neq 0$ has a multiplicative inverse, given by $\frac{1}{a}$. For example, the multiplicative inverse of 3 is $\frac{1}{3}$, since $3 \cdot \frac{1}{3} = 1$, the multiplicative inverse of $\frac{5}{7}$ is $\frac{7}{5}$, since $\frac{5}{7} \cdot \frac{7}{5} = 1$, and so on.

Example 10. Looking at the set $\mathbb{Z}$ of integers, we see that with respect to multiplication the neutral element is the number 1 and we notice that no integer $a \neq 1$ has a multiplicative inverse, since the equation $a \cdot x = 1$ has no integer solutions for $a \neq 1$.

The definition of multiplicative inverse works verbatim for addition as well. In the case of integers, the neutral element with respect to addition is 0, since $a + 0 = 0$ for all integers $a \in \mathbb{Z}$. The additive inverse always exist and is given by the negative number $-a$, since $a + (-a) = 0$.

Example 11. Looking at the set $\mathbb{Z}_6$ of residual classes modulo 6 from example 8, we can use the multiplication table to find multiplicative inverses. To do so, we look at the row of the element and then find the entry equal to 1. If such an entry exists, the element of that column is the multiplicative inverse. If, on the other hand, the row has no entry equal to 1, we know that the element has no multiplicative inverse.

For example in $\mathbb{Z}_6$ the multiplicative inverse of 5 is 5 itself, since $5 \cdot 5 = 1$. We can also see that 5 and 1 are the only elements that have multiplicative inverses in $\mathbb{Z}_6$.

Now, since 5 has a multiplicative inverse modulo 6, we can divide by 5 in $\mathbb{Z}_6$, since we have a notation of multiplicative inverse and division is nothing but multiplication by the multiplicative inverse. For example

$$\frac{4}{5} = 4 \cdot 5^{-1} = 4 \cdot 5 = 2$$

From the last example, we can make the interesting observation that while 5 has no multiplicative inverse as an integer, it has a multiplicative inverse in modular 6 arithmetics.

The remaining question is to understand which elements have multiplicative inverses in modular arithmetics. The answer is that, in modular n arithmetics, a residue class r has a multiplicative inverse, if and only if n and r are coprime. Since $gcd(n, r) = 1$ in that case, we know from the extended Euclidean algorithm that there are numbers s and t, such that

$$1 = s \cdot n + t \cdot r \tag{3.19}$$

If we take the modulus n on both sides, the term $s \cdot n$ vanishes, which tells us that $t \bmod n$ is the multiplicative inverse of r in modular n arithmetics.

Example 12 (Multiplicative inverses in $\mathbb{Z}_6$). In the previous example, we looked up multiplicative inverses in $\mathbb{Z}_6$ from the lookup-table in Example 8. In real world examples, it is usually impossible to write down those lookup tables, as the modulus is way too large, and the sets occasionally contain more elements than there are atoms in the observable universe.

Now, trying to determine that $2 \in \mathbb{Z}_6$ has no multiplicative inverse in $\mathbb{Z}_6$ without using the lookup table, we immediately observe that 2 and 6 are not coprime, since their greatest common divisor is 2. It follows that equation 3.19 has no solutions s and t, which means that 2 has no multiplicative inverse in Z_6.

The same reasoning works for 3 and 4, as neither of these are coprime with 6. The case of 5 is different, since $ggt(6, 5) = 1$. To compute the multiplicative inverse of 5, we use the extended Euclidean algorithm and compute the following:

k	r_k	s_k	$t_k = (r_k - s_k \cdot a) \text{ div } b$
0	6	1	0
1	5	0	1
2	1	1	-1
3	0	.	.

We get $s = 1$ as well as $t = -1$ and have $1 = 1 \cdot 6 - 1 \cdot 5$. From this, it follows that $-1 \bmod 6 = 5$ is the multiplicative inverse of 5 in modular 6 arithmetics. We can double check using Sage:

```
sage: ZZ(6).xgcd(ZZ(5))
(1, 1, -1)
```

At this point, the attentive reader might notice that the situation where the modulus is a prime number is of particular interest, because we know from exercise 8 that in these cases all remainder classes must have modular inverses, since $gcd(r,n) = 1$ for prime n and $r < n$. In fact, Fermat's little theorem provides a way to compute multiplicative inverses in this situation, since in case of a prime modulus p and $r < p$, we get the following:

$$\begin{aligned} r^p &\equiv r \pmod{p} \Leftrightarrow \\ r^{p-1} &\equiv 1 \pmod{p} \Leftrightarrow \\ r \cdot r^{p-2} &\equiv 1 \pmod{p} \end{aligned}$$

This tells us that the multiplicative inverse of a residue class r in modular p arithmetic is precisely r^{p-2}.

Example 13 (Modular 5 arithmetics). To see the unique properties of modular arithmetics when the modulus is a prime number, we will replicate our findings from example 8, but this time for the prime modulus 5. For $n = 5$ we have five equivalence classes of integers which are congruent modulo 5. We write this as follows:

$$\begin{aligned} 0 &:= \{\dots, -5, 0, 5, 10, \dots\} \\ 1 &:= \{\dots, -4, 1, 6, 11, \dots\} \\ 2 &:= \{\dots, -3, 2, 7, 12, \dots\} \\ 3 &:= \{\dots, -2, 3, 8, 13, \dots\} \\ 4 &:= \{\dots, -1, 4, 9, 14, \dots\} \end{aligned}$$

Addition and multiplication can be transferred to the equivalence classes, in a way exactly parallel to Example 8. This results in the following addition and multiplication tables:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Calling the set of remainder classes in modular 5 arithmetics with this addition and multiplication $\mathbb{Z}_5$, we see some subtle but important differences to the situation in $\mathbb{Z}_6$. In particular, we see that in the multiplication table, every remainder $r \neq 0$ has the entry 1 in its row and therefore has a multiplicative inverse. In addition, there are no non-zero elements such that their product is zero.

To use Fermat's little theorem in $\mathbb{Z}_5$ for computing multiplicative inverses (instead of using the multiplication table), let's consider $3 \in \mathbb{Z}_5$. We know that the multiplicative inverse is given by the remainder class that contains $3^{5-2} = 3^3 = 3 \cdot 3 \cdot 3 = 4 \cdot 3 = 2$. And indeed $3^{-1} = 2$, since $3 \cdot 2 = 1$ in $\mathbb{Z}_5$.

We can invoke Sage to do computations in our modular 5 arithmetics type to double-check our computations:

```
sage: Z5 = Integers(5)
sage: Z5(3)**(5-2)
2
sage: Z5(3)**(-1)
2
sage: Z5(3)**(5-2) == Z5(3)**(-1)
True
```

Example 14. To understand one of the principal differences between prime number modular arithmetics and non-prime number modular arithmetics, consider the linear equation $a \cdot x + b = 0$ defined over both types $\mathbb{Z}_5$ and $\mathbb{Z}_6$. Since in $\mathbb{Z}_5$ every non-zero element has a multiplicative inverse, we can always solve these types of equations in $\mathbb{Z}_5$, which is not true in $\mathbb{Z}_6$. To see that, consider the equation $3x + 3 = 0$. In $\mathbb{Z}_5$ we have the following:

$$
\begin{aligned}
3x+3 &= 0 && \text{\# add 2 and on both sides}\\
3x+3+2 &= 2 && \text{\# addition-table: } 2+3=0\\
3x &= 2 && \text{\# divide by 3}\\
2\cdot(3x) &= 2\cdot 2 && \text{\# addition-table: } 2+2=4\\
x &= 4
\end{aligned}
$$

So in the case of our prime number modular arithmetics, we get the unique solution $x = 4$. Now consider $\mathbb{Z}_6$:

$$
\begin{aligned}
3x+3 &= 0 && \text{\# add 3 and on both sides}\\
3x+3+3 &= 3 && \text{\# addition-table: } 3+3=0\\
3x &= 3 && \text{\# no multiplicative inverse of 3 exists}
\end{aligned}
$$

So, in this case, we cannot solve the equation for x by dividing by 3. And, indeed, when we look at the multiplication table of $\mathbb{Z}_6$ (Example 8), we find that there are three solutions $x \in \{1,3,5\}$, such that $3x + 3 = 0$ holds true for all of them.

Exercise 19. Consider the modulus $n = 24$. Which of the integers 7, 1, 0, 805, -4255 have multiplicative inverses in modular 24 arithmetics? Compute the inverses, in case they exist.

Exercise 20. Find the set of all solutions to the congruence $17(2x+5) - 4 \equiv 2x+4 \pmod{5}$. Then project the congruence into $\mathbb{Z}_5$ and solve the resulting equation in $\mathbb{Z}_5$. Compare the results.

Exercise 21. Find the set of all solutions to the congruence $17(2x+5) - 4 \equiv 2x+4 \pmod{6}$. Then project the congruence into $\mathbb{Z}_6$ and try to solve the resulting equation in $\mathbb{Z}_6$.

3.4 Polynomial Arithmetics

A polynomial is an expression consisting of variables (also called indeterminates) and coefficients that involves only the operations of addition, subtraction and multiplication. All coefficients of a polynomial must have the same type, e.g. being integers or rational numbers etc. To be more precise an *univariate polynomial* is an expression

$$P(x) := \sum_{j=0}^{m} a_j x^j = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0 \,, \tag{3.20}$$

where x is called the **indeterminate**, each a_j is called a **coefficient**. If R is the type of the coefficients, then the set of all **univariate**[2] **polynomials with coefficients in** R is written as $R[x]$. We often simply use **polynomial** instead of univariate polynomial, write $P(x) \in R[x]$ for a polynomial and denote the constant term as $P(0)$.

A polynomial is called the **zero polynomial** if all coefficients are zero and a polynomial is called the **one polynomial** if the constant term is 1 and all other coefficients are zero.

Given an univariate polynomial $P(x) = \sum_{j=0}^{m} a_j x^j$ that is not the zero polynomial, we call $deg(P) := m$ the *degree* of P and define the degree of the zero polynomial to be $-\infty$, where $-\infty$ (negative infinity) is a symbol with the properties that $-\infty + m = -\infty$ and $-\infty < m$ for all non-negative integers $m \in \mathbb{N}_0$. In addition, we write

$$Lc(P) := a_m \tag{3.21}$$

and call it the **leading coefficient** of the polynomial P. We can restrict the set $R[x]$ of **all** polynomials with coefficients in R, to the set of all such polynomials that have a degree that does not exceed a certain value. If m is the maximum degree allowed, we write $R_{\leq m}[x]$ for the set of all polynomials with a degree less than or equal to m.

Example 15 (Integer Polynomials). The coefficients of a polynomial must all have the same type. The set of polynomials with integer coefficients is written as $\mathbb{Z}[x]$. Examples of such polynomials are:

$$\begin{aligned}
P_1(x) &= 2x^2 - 4x + 17 && \text{\# with } deg(P_1) = 2 \text{ and } Lc(P_1) = 2 \\
P_2(x) &= x^{23} && \text{\# with } deg(P_2) = 23 \text{ and } Lc(P_2) = 1 \\
P_3(x) &= x && \text{\# with } deg(P_3) = 1 \text{ and } Lc(P_3) = 1 \\
P_4(x) &= 174 && \text{\# with } deg(P_4) = 0 \text{ and } Lc(P_4) = 174 \\
P_5(x) &= 1 && \text{\# with } deg(P_5) = 0 \text{ and } Lc(P_5) = 1 \\
P_6(x) &= 0 && \text{\# with } deg(P_6) = -\infty \text{ and } Lc(P_6) = 0 \\
P_7(x) &= (x-2)(x+3)(x-5)
\end{aligned}$$

In particular, every integer can be seen as an integer polynomial of degree zero. P_7 is a polynomial, because we can expand its definition into $P_7(x) = x^3 - 4x^2 - 11x + 30$, which is a polynomial of degree 3 and leading coefficient 1. The following expressions are not integer polynomials:

$$\begin{aligned}
Q_1(x) &= 2x^2 + 4 + 3x^{-2} \\
Q_2(x) &= 0.5x^4 - 2x \\
Q_3(x) &= 2^x
\end{aligned}$$

In particular Q_1 is not an integer polynomial, because the expression x^{-2} has a negative exponent, Q_2 is not an integer polynomial because the coefficient 0.5 is not an integer and Q_3 is not an integer polynomial because the indeterminant apears in the exponent of of a coefficient.

[2]in our context the term univariate means that the polynomial contains a single variable only

We can invoke Sage to do computations with polynomials. To do so, we have to specify the symbol for the inderteminate and the type for the coefficients (For the definition of rings see 4.2). Note, however that Sage defines the degree of the zero polynomial to be -1.

```
sage: Zx = ZZ['x'] # integer polynomials with indeterminate x
sage: Zt.<t> = ZZ[] # integer polynomials with indeterminate t
sage: Zx
Univariate Polynomial Ring in x over Integer Ring
sage: Zt
Univariate Polynomial Ring in t over Integer Ring
sage: p1 = Zx([17,-4,2])
sage: p1
2*x^2 - 4*x + 17
sage: p1.degree()
2
sage: p1.leading_coefficient()
2
sage: p2 = Zt(t^23)
sage: p2
t^23
sage: p6 = Zx([0])
sage: p6.degree()
-1
```

Example 16 (Polynomials over $\mathbb{Z}_6$). Recall our definition of the remainder classes $\mathbb{Z}_6$ and their arithmetics as defined in example 8. The set of all polynomials with indeterminate x and coefficients in $\mathbb{Z}_6$ is symbolized as $\mathbb{Z}_6[x]$. Example of polynomials from $\mathbb{Z}_6[x]$ are:

$$
\begin{aligned}
&P_1(x) = 2x^2 - 4x + 5 && \text{\# with } deg(P_1) = 2 \text{ and } Lc(P_1) = 2\\
&P_2(x) = x^{23} && \text{\# with } deg(P_2) = 23 \text{ and } Lc(P_2) = 1\\
&P_3(x) = x && \text{\# with } deg(P_3) = 1 \text{ and } Lc(P_3) = 1\\
&P_4(x) = 3 && \text{\# with } deg(P_4) = 0 \text{ and } Lc(P_4) = 3\\
&P_5(x) = 1 && \text{\# with } deg(P_5) = 0 \text{ and } Lc(P_5) = 1\\
&P_6(x) = 0 && \text{\# with } deg(P_5) = -\infty \text{ and } Lc(P_6) = 0\\
&P_7(x) = (x-2)(x+3)(x-5)
\end{aligned}
$$

Just like in the previous example, P_7 is a polynomial. However, since we are working with coefficients from $\mathbb{Z}_6$ now the expansion of P_7 is computed differently, as we have to invoke addition and multiplication in $\mathbb{Z}_6$ as defined in XXX. We get the following:

$$
\begin{aligned}
(x-2)(x+3)(x-5) &= (x+4)(x+3)(x+1) && \text{\# additive inverses in } \mathbb{Z}_6\\
&= (x^2 + 4x + 3x + 3 \cdot 4)(x+1) && \text{\# bracket expansion}\\
&= (x^2 + 1x + 0)(x+1) && \text{\# compuation in } \mathbb{Z}_6\\
&= x^3 + x^2 + x^2 + x && \text{\# bracket expansion}\\
&= x^3 + 2x^2 + x
\end{aligned}
$$

Again, we can use Sage to do computations with polynomials that have their coefficients in $\mathbb{Z}_6$ (For the definition of rings see 4.2). To do so, we have to specify the symbol for the inderteminate and the type for the coefficients:

```
sage: Z6 = Integers(6)
sage: Z6x = Z6['x']
sage: Z6x
Univariate Polynomial Ring in x over Ring of integers modulo 6
sage: p1 = Z6x([5,-4,2])
sage: p1
2*x^2 + 2*x + 5
sage: p1 = Z6x([17,-4,2])
sage: p1
2*x^2 + 2*x + 5
sage: Z6x(x-2)*Z6x(x+3)*Z6x(x-5) == Z6x(x^3 + 2*x^2 + x)
True
```

Given some element from the same type as the coefficients of a polynomial, the polynomial can be evaluated at that element, which means that we insert the given element for every ocurrence of the indeterminate x in the polynomial expression.

To be more precise, let $P \in R[x]$, with $P(x) = \sum_{j=0}^{m} a_j x^j$ be a polynomial with a coefficient of type R and let $b \in R$ be an element of that type. Then the **evaluation** of P at b is given as follows:

$$P(b) = \sum_{j=0}^{m} a_j b^j \tag{3.22}$$

Example 17. Consider the integer polynomials from example 15 again. To evaluate them at given points, we have to insert the point for all occurences of x in the polynomial expression. Inserting arbitrary values from $\mathbb{Z}$, we get:

$$\begin{aligned}
&P_1(2) = 2 \cdot 2^2 - 4 \cdot 2 + 17 = 17 \\
&P_2(3) = 3^{23} = 94143178827 \\
&P_3(-4) = -4 = -4 \\
&P_4(15) = 174 \\
&P_5(0) = 1 \\
&P_6(1274) = 0 \\
&P_7(-6) = (-6-2)(-6+3)(-6-5) = -264
\end{aligned}$$

Note, however, that it is not possible to evaluate any of those polynomial on values of different type. For example, it is not strictly correct to write $P_1(0.5)$, since 0.5 is not an integer. We can verify our computations using Sage:

```
sage: Zx = ZZ['x']
sage: p1 = Zx([17,-4,2])
sage: p7 = Zx(x-2)*Zx(x+3)*Zx(x-5)
sage: p1(ZZ(2))
17
sage: p7(ZZ(-6)) == ZZ(-264)
True
```

Example 18. Consider the polynomials with coefficients in $\mathbb{Z}_6$ from example again. To evaluate them at given values from $\mathbb{Z}_6$, we have to insert the point for all occurences of x in the polynomial expression. We get the following:

$$\begin{aligned}
P_1(2) &= 2\cdot 2^2 - 4\cdot 2 + 5 = 2 - 2 + 5 = 5\\
P_2(3) &= 3^{23} = 3\\
P_3(-4) &= P_3(2) = 2\\
P_5(0) &= 1\\
P_6(4) &= 0
\end{aligned}$$

```
sage: Z6 = Integers(6)
sage: Z6x = Z6['x']
sage: p1 = Z6x([5,-4,2])
sage: p1(Z6(2)) == Z6(5)
True
```

Exercise 22. Compare both expansions of P_7 from $\mathbb{Z}[x]$ and from $\mathbb{Z}_6[x]$ in example 15 and example 18 ,and consider the definition of $\mathbb{Z}_6$ as given in example 8. Can you see how the definition of P_7 over $\mathbb{Z}$ projects to the definition over $\mathbb{Z}_6$ if you consider the residue classes of $\mathbb{Z}_6$?

Polynomial Arithmetics Polynomials behave like integers in many ways. In particular, they can be added, subtracted and multiplied. In addition, they have their own notion of Euclidean division. Informally speaking, we can add two polynomials by simply adding the coefficients of the same index, and we can multiply them by applying the distributive property, that is, by multiplying every term of the left factor with every term of the right factor and adding the results together.

To be more precise let $\sum_{n=0}^{m_1} a_n x^n$ and $\sum_{n=0}^{m_2} b_n x^n$ be two polynomials from $R[x]$. Then the **sum** and the **product** of these polynomials is defined as follows:

$$\sum_{n=0}^{m_1} a_n x^n + \sum_{n=0}^{m_2} b_n x^n = \sum_{n=0}^{max(\{m_1,m_2\})} (a_n + b_n) x^n \tag{3.23}$$

$$\left(\sum_{n=0}^{m_1} a_n x^n\right)\cdot\left(\sum_{n=0}^{m_2} b_n x^n\right) = \sum_{n=0}^{m_1+m_2}\sum_{i=0}^{n} a_i b_{n-i} x^n \tag{3.24}$$

A rule for polynomial subtraction can be deduced from these two rules by first multiplying the subtrahend with (the polynomial) -1 and then add the result to the minuend.

subtrahend

minuend

Regarding the definition of the degree of a polynomial, we see that the degree of the sum is always the maximum of the degrees of both summands, and the degree of the product is always the degree of the sum of the factors, since we defined $-\infty + m = -\infty$ for every integer $m \in \mathbb{Z}$.

Example 19. To given an example of how polynomial arithmetics works, consider the following two integer polynomials $P, Q \in \mathbb{Z}[x]$ with $P(x) = 5x^2 - 4x + 2$ and $Q(x) = x^3 - 2x^2 + 5$. The sum of these two polynomials is computed by adding the coefficients of each term with equal exponent in x. This gives the following:

$$\begin{aligned}
(P+Q)(x) &= (0+1)x^3 + (5-2)x^2 + (-4+0)x + (2+5)\\
&= x^3 + 3x^2 - 4x + 7
\end{aligned}$$

The product of these two polynomials is computed by multiplication of each term in the first factor with each term in the second factor. We get the following:

$$\begin{aligned}(P\cdot Q)(x) &= (5x^2-4x+2)\cdot(x^3-2x^2+5)\\ &= (5x^5-10x^4+25x^2)+(-4x^4+8x^3-20x)+(2x^3-4x^2+10)\\ &= 5x^5-14x^4+10x^3+21x^2-20x+10\end{aligned}$$

```
sage: Zx = ZZ['x']
sage: P = Zx([2,-4,5])
sage: Q = Zx([5,0,-2,1])
sage: P+Q == Zx(x^3 +3*x^2 -4*x +7)
True
sage: P*Q == Zx(5*x^5 -14*x^4 +10*x^3+21*x^2-20*x +10)
True
```

Example 20. Let us consider the polynomials of the previous example but interpreted in modular 6 arithmetics. So we consider $P,Q\in\mathbb{Z}_6[x]$ again with $P(x)=5x^2-4x+2$ and $Q(x)=x^3-2x^2+5$. This time we get the following:

$$\begin{aligned}(P+Q)(x) &= (0+1)x^3+(5-2)x^2+(-4+0)x+(2+5)\\ &= (0+1)x^3+(5+4)x^2+(2+0)x+(2+5)\\ &= x^3+3x^2+2x+1\end{aligned}$$

$$\begin{aligned}(P\cdot Q)(x) &= (5x^2-4x+2)\cdot(x^3-2x^2+5)\\ &= (5x^2+2x+2)\cdot(x^3+4x^2+5)\\ &= (5x^5+2x^4+1x^2)+(2x^4+2x^3+4x)+(2x^3+2x^2+4)\\ &= 5x^5+4x^4+4x^3+3x^2+4x+4\end{aligned}$$

```
sage: Z6x = Integers(6)['x']
sage: P = Z6x([2,-4,5])
sage: Q = Z6x([5,0,-2,1])
sage: P+Q == Z6x(x^3 +3*x^2 +2*x +1)
True
sage: P*Q == Z6x(5*x^5 +4*x^4 +4*x^3+3*x^2+4*x +4)
True
```

Exercise 23. Compare the sum $P+Q$ and the product $P\cdot Q$ from the previous two examples 19 and 20 and consider the definition of $\mathbb{Z}_6$ as given in example 8. How can we derive the computations in $\mathbb{Z}_6[x]$ from the computations in $Z[x]$?

Euklidean Division The arithmetics of polynomials share a lot of properties with the arithmetics of integers and as a consequence the concept of Euclidean division and the algorithm of long division is also defined for polynomials. Recalling the Euclidean division of integers 3.2, we know that, given two integers a and $b \neq 0$, there is always another integer m and a natural number r with $r < |b|$ such that $a = m \cdot b + r$ holds.

We can generalize this to polynomials whenever the leading coefficient of the dividend polynomial has a notion of multiplicative inverse. In fact, given two polynomials A and $B \neq 0$ from $R[x]$ such that $Lc(B)^{-1}$ exists in R, there exist two polynomials Q (the quotient) and P (the remainder), such that the following equation holds:

$$A = Q \cdot B + P \tag{3.25}$$

and $deg(P) < deg(B)$. Similarly to integer Euclidean division, both Q and P are uniquely defined by these relations.

Notation and Symbols 2. Suppose that the polynomials A, B, Q and P satisfy equation 3.25. We often use the following notation to describe the quotient and the remainder polynomials of the Euclidean division:

$$A \text{ div } B := Q, \qquad A \text{ mod } B := P \tag{3.26}$$

We also say that a polynomial A is divisible by another polynomial B if A mod $B = 0$ holds. In this case, we also write $B|A$ and call B a *factor* of A.

Analogously to integers, methods to compute Euclidean division for polynomials are called **polynomial division algorithms**. Probably the best known algorithm is the so called **polynomial long division** .

algorithm-floating

Algorithm 3 Polynomial Euclidean Algorithm

Require: $A, B \in R[x]$ with $B \neq 0$, such that $Lc(B)^{-1}$ exists in R
  procedure POLY-LONG-DIVISION(A, B)
    $Q \leftarrow 0$
    $P \leftarrow A$
    $d \leftarrow deg(B)$
    $c \leftarrow Lc(B)$
    while $deg(P) \geq d$ **do**
      $S := Lc(P) \cdot c^{-1} \cdot x^{deg(P)-d}$
      $Q \leftarrow Q + S$
      $P \leftarrow P - S \cdot B$
    end while
    return (Q, P)
  end procedure
Ensure: $A = Q \cdot B + P$

This algorithm works only when there is a notion of division by the leading coefficient of B. It can be generalized, but we will only need this somewhat simpler method in what follows.

Example 21 (Polynomial Long Division). To give an example of how the previous algorithm works, let us divide the integer polynomial $A(x) = x^5 + 2x^3 - 9 \in \mathbb{Z}[x]$ by the integer polynomial $B(x) = x^2 + 4x - 1 \in \mathbb{Z}[x]$. Since B is not the zero polynomial and the leading coefficient of B is 1, which is invertible as an integer, we can apply algorithm 1. Our goal is to find solutions to equation XXX, that is, we need to find the quotient polynomial $Q \in \mathbb{Z}[x]$ and the reminder polynomial $P \in \mathbb{Z}[x]$ such that $x^5 + 2x^3 - 9 = Q(x) \cdot (x^2 + 4x - 1) + P(x)$. Using a notation that is mostly used in anglophone countries, we compute as follows:

$$
\begin{array}{r|rrrrrr}
\multicolumn{1}{r}{} & & & X^3 & -4X^2 & +19X & -80 \\ \cline{2-7}
X^2+4X-1 & X^5 & & +2X^3 & & & -9 \\
 & -X^5 & -4X^4 & +X^3 & & & \\ \cline{2-4}
 & & -4X^4 & +3X^3 & & & \\
 & & 4X^4 & +16X^3 & -4X^2 & & \\ \cline{3-5}
 & & & 19X^3 & -4X^2 & & \\
 & & & -19X^3 & -76X^2 & +19X & \\ \cline{4-6}
 & & & & -80X^2 & +19X & -9 \\
 & & & & 80X^2 & +320X & -80 \\ \cline{5-7}
 & & & & & 339X & -89
\end{array} \tag{3.27}
$$

We therefore get $Q(x) = x^3 - 4x^2 + 19x - 80$ as well as $P(x) = 339x - 89$ and indeed we have $x^5 + 2x^3 - 9 = (x^3 - 4x^2 + 19x - 80) \cdot (x^2 + 4x - 1) + (339x - 89)$, which we can double check invoking Sage:

```
sage: Zx = ZZ['x']
sage: A = Zx([-9,0,0,2,0,1])
sage: B = Zx([-1,4,1])
sage: Q = Zx([-80,19,-4,1])
sage: P = Zx([-89,339])
sage: A == Q*B + P
True
```

Example 22. In the previous example, polynomial division gave a non-trivial (non-vanishing, i.e non-zero) remainder. Of special interest are divisions that don't give a remainder. Such divisors are called factors of the dividend.

For example, consider the integer polynomial P_7 from example 15 again. As we have shown, it can be written both as $x^3 - 4x^2 - 11x + 30$ and as $(x-2)(x+3)(x-5)$. From this, we can see that the polynomials $F_1(x) = (x-2)$, $F_2(x) = (x+3)$ and $F_3(x) = (x-5)$ are all factors of $x^3 - 4x^2 - 11x + 30$, since division of P_7 by any of these factors will result in a zero remainder.

Exercise 24. Consider the polynomial expressions $A(x) := -3x^4 + 4x^3 + 2x^2 + 4$ and $B(x) = x^2 - 4x + 2$. Compute the Euclidean division of A by B in the following types:

1. $A, B \in \mathbb{Z}[x]$
2. $A, B \in \mathbb{Z}_6[x]$
3. $A, B \in \mathbb{Z}_5[x]$

Now consider the result in $\mathbb{Z}[x]$ and in $\mathbb{Z}_6[x]$. How can we compute the result in $\mathbb{Z}_6[x]$ from the result in $\mathbb{Z}[x]$?

Exercise 25. Show that the polynomial $B(x) = 2x^4 - 3x + 4 \in \mathbb{Z}_5[x]$ is a factor of the polynomial $A(x) = x^7 + 4x^6 + 4x^5 + x^3 + 2x^2 + 2x + 3 \in \mathbb{Z}_5[x]$ that is show $B|A$. What is B div A?

Prime Factors Recall that the fundamental theorem of arithmetics 3.6 tells us that every natural number is the product of prime numbers. In this chapter we will see that something similar holds for univariate polynomials $R[x]$, too[3].

[3] Strictly speaking this is not true for polynomials over arbitrary types R. However in this book we assume R to be a so called unique factorization domain for which the content of this section holds.

The polynomial analog to a prime number is a so called an **irreducible polynomial**, which is defined as a polynomial that cannot be factored into the product of two non-constant polynomials using Euclidean division. Irreducible polynomials are for polynomials what prime numbers are for integer: They are the basic building blocks from which all other polynomials can be constructed. To be more precise, let $P \in R[x]$ be any polynomial. Then there are always irreducible polynomials $F_1, F_2, \ldots, F_k \in R[x]$, such that the following holds:

$$P = F_1 \cdot F_2 \cdot \ldots \cdot F_k \,. \tag{3.28}$$

This representation is unique, except for permutations in the factors and is called the **prime factorization** of P. Moreover each factor F_i is called a **prime factor** of P.

Example 23. Consider the polynomial expression $P = x^2 - 3$. When we interpret P as an integer polynomial $P \in \mathbb{Z}[x]$, we find that this polynomial is irreducible, since any factorization other then $1 \cdot (x^2 - 3)$, must look like $(x-a)(x+a)$ for some integer a, but there is no integers a with $a^2 = 3$.

```
sage: Zx = ZZ['x']
sage: p = Zx(x^2-3)
sage: p.factor()
x^2 - 3
```

On the other hand interpreting P as a polynomial $P \in \mathbb{Z}_6[x]$ in modulo 6 arithmetics, we see that P has two factors $F_1 = (x-3)$ and $F_2 = (x+3)$, since $(x-3)(x+3) = x^2 - 3x + 3x - 3 \cdot 3 = x^2 - 3$.

Points where a polynomial evaluates to zero are called **roots** of the polynomial. To be more precise, let $P \in R[x]$ be a polynomial. Then a root is a points $x_0 \in R$ with $P(x_0) = 0$ and the set of all roots of P is defined as follows:

$$R_0(P) := \{x_0 \in R \mid P(x_0) = 0\} \tag{3.29}$$

The roots of a polynomial are of special interest with respect to it's prime factorization, since it can be shown that for any given root x_0 of P the polynomial $F(x) = (x - x_0)$ is a prime factor of P.

Finding the roots of a polynomial is sometimes called **solving the polynomial**. It is a hard problem and has been the subject of much research throughout history.

It can be shown that if m is the degree of a polynomial P, then P can not have more than m roots. However, in general, polynomials can have less than m roots.

Example 24. Consider the integer polynomial $P_7(x) = x^3 - 4x^2 - 11x + 30$ from example 15 again. We know that its set of roots is given by $R_0(P_7) = \{-3, 2, 5\}$.

On the other hand, we know from example 23 that the integer polynomial $x^2 - 3$ is irreducible. It follows that it has no roots, since every root defines a prime factor.

Example 25. To give another example, consider the integer polynomial $P = x^7 + 3x^6 + 3x^5 + x^4 - x^3 - 3x^2 - 3x - 1$. We can invoke Sage to compute the roots and prime factors of P:

```
sage: Zx = ZZ['x']
sage: p = Zx(x^7 + 3*x^6 + 3*x^5 + x^4 - x^3 - 3*x^2 - 3*x - 1
   )
sage: p.roots()
[(1, 1), (-1, 4)]
```

```
sage: p.factor()
(x - 1) * (x + 1)^4 * (x^2 + 1)
```

We see that P has the root 1 and that the associated prime factor $(x-1)$ occurs once in P and that it has the root -1, where the associated prime factor $(x+1)$ occurs 4 times in P. This gives the following prime factorization:

$$P = (x-1)(x+1)^4(x^2+1)$$

Exercise 26. Show that if a polynomial $P \in R[x]$ of degree $deg(P) = m$ has less then m roots, it must have a prime factor F of degree $deg(F) > 1$.

Exercise 27. Consider the polynomial $P = x^7 + 3x^6 + 3x^5 + x^4 - x^3 - 3x^2 - 3x - 1 \in \mathbb{Z}_6[x]$. Compute the set of all roots of $R_0(P)$ and then compute the prime factorization of P.

Lagrange interpolation One particularly useful property of polynomials is that a polynomial of degree m is completely determined on $m+1$ evaluation points. Seeing this from a different angle, we can (sometimes) uniquely derive a polynomial of degree m from a set S:

$$S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m) \mid x_i \neq x_j \text{ for all indices i and j}\} \tag{3.30}$$

Polynomials therefore have the property that $m+1$ pairs of points (x_i, y_i) are enough to determine the set of pairs $(x, P(x))$ for all $x \in R$. This “few too many” property of polynomials is used in many places, like for example in erasure codes. It is also of importance in snarks and we therefore need to understand a method to actually compute a polynomial from a set of points.

If the coefficients of the polynomial we want to find have a notion of multiplicative inverse, it is always possible to find such a polynomial. One method for this is called **Lagrange interpolation**. It works as follows: Given a set like 3.30, a polynomial P of degree m with $P(x_i) = y_i$ for all pairs (x_i, y_i) from S is given by the following algorithm:

Algorithm 4 Lagrange Interpolation

Require: R must have multiplicative inverses
Require: $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m) \mid x_i, y_i \in R, x_i \neq x_j \text{ for all indices i and j}\}$
  procedure LAGRANGE-INTERPOLATION(S)
    for $j \in (0 \ldots m)$ **do**
      $l_j(x) \leftarrow \Pi_{i=0;i\neq j}^m \frac{x-x_i}{x_j-x_i} = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_m)}{(x_j-x_m)}$
    end for
    $P \leftarrow \sum_{j=0}^m y_j \cdot l_j$
    return P
  end procedure
Ensure: $P \in R[x]$ with $deg(P) = m$
Ensure: $P(x_j) = y_j$ for all pairs $(x_j, y_j) \in S$

Example 26. Let us consider the set $S = \{(0,4), (-2,1), (2,3)\}$. Our task is to compute a polynomial of degree 2 in $\mathbb{Q}[x]$ with coefficients from the rational numbers $\mathbb{Q}$. Since $\mathbb{Q}$ has multiplicative inverses, we can use the Lagrange interpolation algorithm from 4, to compute the polynomial.

$$\begin{aligned}
l_0(x) &= \frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2} = \frac{x+2}{0+2}\cdot\frac{x-2}{0-2} = -\frac{(x+2)(x-2)}{4}\\
&= -\frac{1}{4}(x^2-4)\\
l_1(x) &= \frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2} = \frac{x-0}{-2-0}\cdot\frac{x-2}{-2-2} = \frac{x(x-2)}{8}\\
&= \frac{1}{8}(x^2-2x)\\
l_2(x) &= \frac{x-x_0}{x_2-x_0}\cdot\frac{x-x_1}{x_2-x_1} = \frac{x-0}{2-0}\cdot\frac{x+2}{2+2} = \frac{x(x+2)}{8}\\
&= \frac{1}{8}(x^2+2x)\\
P(x) &= 4\cdot(-\frac{1}{4}(x^2-4)) + 1\cdot\frac{1}{8}(x^2-2x) + 3\cdot\frac{1}{8}(x^2+2x)\\
&= -x^2+4+\frac{1}{8}x^2-\frac{1}{4}x+\frac{3}{8}x^2+\frac{3}{4}x\\
&= -\frac{1}{2}x^2+\frac{1}{2}x+4
\end{aligned}$$

And, indeed, evaluation of P on the x-values of S gives the correct points, since $P(0) = 4$, $P(-2) = 1$ and $P(2) = 3$.

```
sage: Qx = QQ['x']
sage: S=[(0,4),(-2,1),(2,3)]
sage: Qx.lagrange_polynomial(S)
-1/2*x^2 + 1/2*x + 4
```

Example 27. To give another example more relevant to the topics of this book, let us consider the same set $S = \{(0,4),(-2,1),(2,3)\}$ as in the previous example. This time, the task is to compute a polynomial $P \in \mathbb{Z}_5[x]$ from this data. Since we know from example 13 that multiplicative inverses exist in $\mathbb{Z}_5$, algorithm 4 applies and we can compute a unique polynomial of degree 2 in $\mathbb{Z}_5[x]$ from S. We can use the lookup tables from example 13 for computation in $\mathbb{Z}_5$ and get the following:

$$\begin{aligned}
l_0(x) &= \frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2} = \frac{x+2}{0+2}\cdot\frac{x-2}{0-2} = \frac{(x+2)(x-2)}{-4} = \frac{(x+2)(x+3)}{1}\\
&= x^2+1\\
l_1(x) &= \frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2} = \frac{x-0}{-2-0}\cdot\frac{x-2}{-2-2} = \frac{x}{3}\cdot\frac{x+3}{1} = 2(x^2+3x)\\
&= 2x^2+x\\
l_2(x) &= \frac{x-x_0}{x_2-x_0}\cdot\frac{x-x_1}{x_2-x_1} = \frac{x-0}{2-0}\cdot\frac{x+2}{2+2} = \frac{x(x+2)}{3} = 2(x^2+2x)\\
&= 2x^2+4x\\
P(x) &= 4\cdot(x^2+1)+1\cdot(2x^2+x)+3\cdot(2x^2+4x)\\
&= 4x^2+4+2x^2+x+x^2+2x\\
&= 2x^2+3x+4
\end{aligned}$$

And, indeed, evaluation of P on the x-values of S gives the correct points, since $P(0) = 4$, $P(-2) = 1$ and $P(2) = 3$.

```
sage: F5 = GF(5)
sage: F5x = F5['x']
sage: S=[(0,4),(-2,1),(2,3)]
sage: F5x.lagrange_polynomial(S)
2*x^2 + 3*x + 4
```

Exercise 28. Consider modular 5 arithmetics from example 13 and the set $S = \{(0,0),(1,1),(2,2),(3,2)\}$. Find a polynomial $P \in \mathbb{Z}_5[x]$ such that $P(x_i) = y_i$ for all $(x_i, y_i) \in S$.

Exercise 29. Consider the set S from the previous example. Why is it not possible to apply algorithm 4 to construct a polynomial $P \in \mathbb{Z}_6[x]$, such that $P(x_i) = y_i$ for all $(x_i, y_i) \in S$?

Chapter 4

Algebra

Def Subgroup, Fundamental theorem of cyclic groups.

In the previous chapter, we gave an introduction to the basic computational skills needed for a pen-and-paper approach to SNARKs. This chapter provides a more abstract clarification of relevant mathematical terminology on **algebraic types** such as **groups**, **fields**, **rings** and similar.

In a nutshell, algebraic types define sets that are analogous to numbers in various aspects, in the sense that you can add, subtract, multiply or divide on those sets. We know many examples of sets that fall under those categories, such as natural numbers, integers, rational or the real numbers. In some sense, these are the most fundamental examples of such sets.

Papers on cryptography (and mathematical papers in general) frequently contain such terms, and it is necessary to get at least some understanding of these terms to be able to follow these papers. In this chapter, we therefore provide a short introduction to these concepts.

4.1 Groups

Groups are abstractions that capture the essence of mathematical phenomena, like addition and subtraction, multiplication and division, permutations, or symmetries.

To understand groups, let us think back to when we learned about the addition and subtraction of integers (also called whole numbers) in school. We have learned that, whenever we add two integers, the result is guaranteed to be an integer as well. We have also learned that adding zero to any integer means that "nothing happens", that is, the result of the addition is the same integer we started with. Furthermore, we have learned that the order in which we add two (or more) integers does not matter, that brackets have no influence on the result of addition, and that, for every integer, there is always another integer (the negative) such that we get zero when we add them together.

These conditions are the defining properties of a group, and mathematicians have recognized that the exact same set of rules can be found in very different mathematical structures. It therefore makes sense to give a formal definition of what a group should be, detached from any concrete examples. This lets us handle entities of very different mathematical origins in a flexible way, while retaining essential structural aspects of many objects in abstract algebra and beyond.

Distilling these rules to the smallest independent list of properties and making them abstract, we arrive at the definition of a group:

Definition 4.1.0.1. A **group** $(\mathbb{G},\cdot)$ is a set $\mathbb{G}$, together with a **map** $\cdot$. The map, also denoted as $\mathbb{G}\times\mathbb{G}\rightarrow\mathbb{G}$ and called the **group law**, combines two elements of the set $\mathbb{G}$ into a third one such that the following properties hold:

- **Existence of a neutral element**: There is a $e \in \mathbb{G}$ for all $g \in \mathbb{G}$, such that $e \cdot g = g$ as well as $g \cdot e = g$.

- **Existence of an inverse**: For every $g \in \mathbb{G}$ there is a $g^{-1} \in \mathbb{G}$, such that $g \cdot g^{-1} = e$ as well as $g^{-1} \cdot g = e$.

- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.

Rephrasing the abstract definition in layman's terms, a group is something where we can do computations in a way that resembles the behavior of the addition of integers. Specifically, this means we can combine some element with another element into a new element in a way that is reversible and where the order of combining elements doesn't matter.

Notation and Symbols 3. Let $(\mathbb{G}\cdot)$ be a finite group. If there is no risk of ambiguity (about what the group law of that group is), we frequently drop the symbol $\cdot$ and simply write $\mathbb{G}$ as the notation for the group, keeping the group law implicit.

As we will see in XXX, groups are heavily used in cryptography and in SNARKs. But let us look at some more familiar examples fist:

add reference

Example 28 (Integer Addition and Subtraction). The set $(\mathbb{Z}, +)$ of integers with integer addition is the archetypical example of a group, where the group law is traditionally written as $+$ (instead of $\cdot$). To compare integer addition against the abstract axioms of a group, we first see that the neutral element e is the number 0, since $a + 0 = a$ for all integers $a \in \mathbb{Z}$. Furthermore, the inverse of a number is its negative counterpart, since $a + (-a) = 0$, for all $a \in \mathbb{Z}$. In addition, we know that $(a+b)+c = a+(b+c)$, so integers with addition are indeed a group in the abstract sense.

Example 29 (The trivial group). The most basic example of a group is group with just one element $\{\bullet\}$ and the group law $\bullet \cdot \bullet = \bullet$.

Add real-life example of 0?

Commutative Groups When we look at the general definition of a group, we see that it is somewhat different from what we know from integers. We know that the order in which we add two integers doesn't matter, as, for example, $4+2$ is the same as $2+4$. However, we also know from example XXX that this is not the case for all groups.

add reference

This means that groups where the order in which the group law is executed doesn't matter are a special subcase of groups called **commutative group**s. To be more precise, a group is called commutative if $g_1 \cdot g_2 = g_2 \cdot g_1$ holds for all $g_1, g_2 \in \mathbb{G}$.

Notation and Symbols 4. For commutative groups $(\mathbb{G}, \cdot)$, we frequently use the so-called **additive notation** $(\mathbb{G}, +)$, that is, we write $+$ instead of $\cdot$ for the group law, and $-g := g^{-1}$ for the inverse of an element $g \in \mathbb{G}$.

Example 30. Consider the group of integers with integer addition again. Since $a+b = b+a$ for all integers, this group is the archetypical example of a commutative group. Since there are infinitely many integers, $(\mathbb{Z}, +)$ is not a finite group.

Example 31. Consider our definition of modulo 6 residue classes $(\mathbb{Z}_6, +)$ as defined in the addition table from example 8. As we can see, the residue class 0 is the neutral element in modulo 6 arithmetics, and the inverse of a residue class r is given by $6-r$, since $r + (6-r) = 6$, which is congruent to 0, since $6 \bmod 6 = 0$. Moreover, $(r_1 + r_2) + r_3 = r_1 + (r_2 + r_3)$ is inherited from integer arithmetic.

We therefore see that $(\mathbb{Z}_6, +)$ is a group, and, since the addition table in example 8 is symmetrical, we see $r_1 + r_2 = r_2 + r_1$, which shows that $(\mathbb{Z}_6, +)$ is commutative.

The previous example of a commutative group is a very important one for this book. Abstracting from this example and considering residue classes $(\mathbb{Z}_n,+)$ for arbitrary moduli n, it can be shown that $(\mathbb{Z},+)$ is a commutative group with the neutral element 0 and the additive inverse $n-r$ for any element $r \in \mathbb{Z}_n$. We call such a group the **remainder class group** of modulus n.

Of particular importance for pairing-based cryptography in general and SNARKs in particular are so-called **pairing maps** on commutative groups. To be more precise let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ be three commutative groups. For historical reasons, we write the group law on $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation and the group law on $\mathbb{G}_3$ in multiplicative notation. Then a **pairing map** is a function

$$e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3 \tag{4.1}$$

This function takes pairs (g_1,g_2) (products) of elements from $\mathbb{G}_1$ and $\mathbb{G}_2$, and maps them to elements from $\mathbb{G}_3$, such that the **bilinearity** property holds:

Definition 4.1.0.2. **Bilinearity**
For all $g_1,g_1' \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ we have $e(g_1+g_1',g_2) = e(g_1,g_2)\cdot e(g_1',g_2)$ and for all $g_1 \in \mathbb{G}_1$ and $g_2,g_2' \in \mathbb{G}_2$ we have $e(g_1,g_2+g_2') = e(g_1,g_2)\cdot e(g_1,g_2')$.

A pairing map is called **non-degenerate** if, whenever the result of the pairing is the neutral element in $\mathbb{G}_3$, one of the input values is the neutral element of $\mathbb{G}_1$ or $\mathbb{G}_2$. To be more precise, $e(g_1,g_2) = e_{\mathbb{G}_3}$ implies $g_1 = e_{\mathbb{G}_1}$ or $g_2 = e_{\mathbb{G}_2}$.

Informally speaking, bilinearity means that it doesn't matter if we first execute the group law on one side and then apply the bilinear map, or if we first apply the bilinear map and then apply the group law. Moreover, non-degeneracy means that the result of the pairing is zero if and only if at least one of the input values is zero.

Example 32. One of the most basic examples of a non-degenerate pairing involves $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ all to be the groups of integers with addition $(\mathbb{Z},+)$. Then the following map defines a non-degenerate pairing:

$$e(\cdot,\cdot) : \mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}\ (a,b) \mapsto a\cdot b$$

Note that bilinearity follows from the distributive law of integers, since for $a,b,c \in \mathbb{Z}$, we have $e(a+b,c) = (a+b)\cdot c = a\cdot c + b\cdot c = e(a,c)+e(b,c)$ and the same reasoning is true for the second argument.

To the see that $e(\cdot,\cdot)$ is non-degenrate, assume that $e(a,b) = 0$. Then a$\cdot b = 0$ implies that a or b must be zero.

Exercise 30. Consider example 13 again and let $\mathbb{Z}_5^*$ be the set of all remainder classes from $\mathbb{Z}_5$ without the class 0. Then $\mathbb{Z}_5^* = \{1,2,3,4\}$. Show that $(\mathbb{Z}_5^*,\cdot)$ is a commutative group. check reference

Exercise 31. Generalizing the previous exercise, consider the general modulus n, and let $\mathbb{Z}_n^*$ be the set of all remainder classes from $\mathbb{Z}_n$ without the class 0. Then $\mathbb{Z}_n^* = \{1,2,\ldots,n-1\}$. Provide a counter-example to show that $(\mathbb{Z}_n^*,\cdot)$ is not a group in general.

Find a condition such that $(\mathbb{Z}_n^*,\cdot)$ is a commutative group, compute the neutral element, give a closed form for the inverse of any element and prove the commutative group axioms.

Exercise 32. Consider the remainder class groups $(\mathbb{Z}_n,+)$ for some modulus n. Show that the map

$$e(\cdot,\cdot) : \mathbb{Z}_n\times\mathbb{Z}_n \to \mathbb{Z}_n\ (a,b) \mapsto a\cdot b$$

is bilinear. Why is it not a pairing in general and what condition must be imposed on n, such that the map will be a pairing?

Finite groups As we have seen in the previous examples, groups can either contain infinitely many elements (such as integers) or finitely many elements (as for example the remainder class groups $(\mathbb{Z}_n,+)$). To capture this distinction, a group is called a **finite group** if the underlying set of elements is finite. In that case, the number of elements of that group is called its **order**.

Notation and Symbols 5. Let $\mathbb{G}$ be a finite group. We write $ord(\mathbb{G})$ or $|\mathbb{G}|$ for the order of $\mathbb{G}$.

Example 33. Consider the remainder class groups $(\mathbb{Z}_6,+)$ from example 8 and $(\mathbb{Z}_5,+)$ from example 13, and the group $(\mathbb{Z}_5^*,\cdot)$ from exercise 30. We can easily see that the order of $(\mathbb{Z}_6,+)$ is 6, the order of $(\mathbb{Z}_5,+)$ is 5 and the order of $(\mathbb{Z}_5^*,\cdot)$ is 4.

check references to previous examples

To be more general, considering arbitrary moduli n, we know from Euclidean division that the order of the remainder class group $(\mathbb{Z}_n,+)$ is n.

Exercise 33. The RSA crypto system is based on a modulus n that is typically the product of two prime numbers of size 2048-bits. What is the approximate order of the remainder class group $(\mathbb{Z}_n,+)$ in this case?

RSA crypto system

size 2048-bits

Generators These are sets of elements that can be used to generate the entire group by applying the group law repeatedly to these elements or their inverses only. Generators are of particular interest when working with groups.

Of course, every group $\mathbb{G}$ has a trivial set of generators, when we just consider every element of the group to be in the generator set. The more interesting question is to find the smallest possible set of generators for a given group. Of particular interest in this regard are groups that have a single generator, that is, there exists an element $g \in \mathbb{G}$ such that every other element from $\mathbb{G}$ can be computed by the repeated combination of g and its inverse g^{-1} only. Groups with a single generator are called **cyclic groups**.

Example 34. The most basic example of a cyclic group is the group of integers $(\mathbb{Z},+)$ with integer addition. 1 is a single generator of $\mathbb{Z}$, since every integer can be obtained by repeatedly adding either 1 or its inverse -1 to itself. For example -4 is generated by -1, since $-4 = -1+(-1)+(-1)+(-1)$.

Example 35. Consider a modulus n and the remainder class groups $(\mathbb{Z}_n,+)$ from example 33. These groups are cyclic, with a generator 1, since every other element of that group can be constructed by repeatedly adding the remainder class 1 to itself. Since $\mathbb{Z}_n$ is also finite, we know that $(\mathbb{Z}_n,+)$ is a finite cyclic group of order n.

check reference

Example 36. Let $p \in \mathbb{P}$ be prime number and $(\mathbb{F}_p^*,\cdot)$ the finite group from exercise XXX. Then $(\mathbb{F}_p^*,\cdot)$ is cyclic and every element $g \in \mathbb{F}_q^*$ is a generator.

add reference: 31?

The discrete Logarithm problem Observe that, when $\mathbb{G}$ is a cyclic group of order n and $g \in \mathbb{G}$ is a generator of $\mathbb{G}$, then there is a map with respect to the generator g with the following properties:

$$g^{(\cdot)} : \mathbb{Z}_n \to \mathbb{G}\; x \mapsto g^x \tag{4.2}$$

In the map above, g^x means "multiply g x-times by itself" and $g^0 = e_{\mathbb{G}}$. This map, called the **exponential map**, has the remarkable property that it maps the additive group law of the remainder class group $(\mathbb{Z}_n,+)$ in a one-to-one correspondence to the group law of $\mathbb{G}$.

To see this, first observe that, since $g^0 := e_{\mathbb{G}}$ by definition, the neutral element of $\mathbb{Z}_n$ is mapped to the neutral element of $\mathbb{G}$, and, since $g^{x+y} = g^x \cdot g^y$, the map respects the group law.

Because the exponential map respects the group law, it doesn't matter if we do our computation in $\mathbb{Z}_n$ before we write the result into the exponent of g or afterwards: the result will be the same in both cases. This is usually referred to as doing computations "in the exponent". In cryptography in general, and in SNARK development in particular, we often perform computations "in the exponent" of a generator.

Example 37. Consider the multiplicative group $(\mathbb{F}_5^*, \cdot)$ from example 30. We know that $\mathbb{F}_5^*$ is a cyclic group of order 4, and that every element is a generator. If we choose $3 \in \mathbb{Z}_5^*$, we then know that the following map respects the group law of addition in $\mathbb{Z}_4$ and the group law of multiplication in $\mathbb{Z}_5^*$:

check reference

$$3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{Z}_5^* \; x \mapsto 3^x$$

Let us now perform a computation in the exponent:

$$\begin{aligned} 3^{2+3-2} &= 3^3 \\ &= 2 \end{aligned}$$

This gives the same result as doing the same computation in $\mathbb{F}*_5$:

$$\begin{aligned} 3^{2+3-2} &= 3^2 \cdot 3^3 \cdot 3^{-2} \\ &= 4 \cdot 2 \cdot (-3)^2 \\ &= 3 \cdot 2^2 \\ &= 3 \cdot 4 \\ &= 2 \end{aligned}$$

Since the exponential map is a one-to-one correspondence that respects the group law, it can be shown that this map has an inverse with respect to the base g, called the **base g discrete logarithm map**:

$$log_g(\cdot) : \mathbb{G} \to \mathbb{Z}_n \; x \mapsto log_g(x) \tag{4.3}$$

Discrete logarithms are highly important in cryptography, because there are groups such that the exponential map and its inverse, the discrete logarithm, which are believed to be one-way functions, that is, while it is possible to compute the exponential map in polynomial time, computing the discrete log takes (sub)-exponential time. We have discussed this briefly following example 3.5 in the previous chapter, and will look at this and similar problems in more detail in the next section.

polynomial time

exponential time

TODO: Fundamental theorem of finite cyclic groups

4.1.1 Cryptographic Groups

In this section, we will look at families of groups that are believed to satisfy certain **computational hardness assumptions**, namely that a particular problem cannot be solved efficiently (where efficiently typically means "in polynomial time of a given security parameter") in the groups under consideration.

Example 38. To highlight the concept of the computational hardness assumption, consider the group of integers $\mathbb{Z}$ from example 3.5. One of the best known and most researched examples of computational hardness is the assumption that the factorization of integers into prime numbers cannot be solved by any algorithm in polynomial time with respect to the bit-length of the integer.

check reference

To be more precise, the computational hardness assumption of integer factorization assumes that, given any integer $z \in \mathbb{Z}$ with bit-length b, there is no integer k and no algorithm with the runtime complexity of $\mathcal{O}(b^k)$ that is able to find the prime numbers $p_1, p_2, \ldots, p_j \in \mathbb{P}$, such that $z = p_1 \cdot p_2 \cdot \ldots \cdot p_j$.

runtime complexity

This hardness assumption was proven to be false, since Shor's (**?**) algorithm shows that integer factorization is at least efficiently possible on a quantum computer, since the runtime complexity of this algorithm is $\mathcal{O}(b^3)$. However, no such algorithm is known on a classical computer.

S: what does "efficiently" mean here?

In the realm of classical computers, however, we still have to call the non-existence of such an algorithm an "assumption" because, to date, there is no proof that it is actually impossible to find one. The problem is that it is hard to reason about algorithms that we don't know.

So, despite the fact that there is currently no known algorithm that can factor integers efficiently on a classical computer, we cannot exclude that such an algorithm might exist in principle, and that someone eventually will discover it in the future.

However, what still makes the assumption plausible, despite the absence of any actual proof, is the fact that, after decades of extensive search, still no such algorithm has been found.

In what follows, we will describe a few computational hardness assumptions that arise in the context of groups in cryptography, because we will refer to them throughout the book.

computational hardness assumptions

The discrete logarithm assumption The so-called discrete logarithm problem is one of the most fundamental assumptions in cryptography. To define it, let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. We know from 4.2 that there is an exponential map $g^{(\cdot)} : \mathbb{Z}_r \to \mathbb{G} : x \mapsto g^x$ that maps the residue classes from modulo r arithmetic onto the group in a $1:1$ correspondence. The **discrete logarithm problem** is the task of finding inverses to this map, that is, to find a solution $x \in \mathbb{Z}_r$ to the following equation for some given $h \in \mathbb{G}$:

check reference

$$h = g^x \tag{4.4}$$

In other words, the **discrete logarithm assumption (DL-A)** is the assumption that there exists no algorithm with polynomial running time in the security parameter $log_2(r)$, that is able to compute some x if only h, g and g^x are given in $\mathbb{G}$. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a **DL-A group**.

Rephrasing the previous definition, DL-A groups are believed to have the property that it is infeasible to compute some number x that solves the equation $h = g^x$ for a given h and g, assuming that the size of the group r is large enough.

Example 39 (Public key cryptography). One the most basic examples of an application for DL-A groups is in public key cryptography, where the parties publicly agree on some pair $(\mathbb{G}, g)$ such that $\mathbb{G}$ is a finite cyclic group of sufficiently large order r, where $\mathbb{G}$ is believed to be a DL-A group, and g is a generator of $\mathbb{G}$.

In this setting, a secret key is some number $sk \in \mathbb{Z}_r$ and the associated public key pk is the group element $pk = g^{sk}$. Since discrete logarithms are assumed to be hard, it is infeasible for an attacker to compute the secret key from the public key, since it is believed to be hard to find solutions x to the following equation:

$$pk = g^x \tag{4.5}$$

As the previous example shows, identifying DL-A groups is an important practical problem. Unfortunately, it is easy to see that it does not make sense to assume the hardness of the discrete logarithm problem in all finite cyclic groups: Counterexamples are common and easy to construct.

Example 40 (Modular arithmetics for Fermat's primes). It is widely believed that the discrete logarithm problem is hard in multiplicative groups $\mathbb{Z}_p^*$ of prime number modular arithmetics. However, this is not true in general. To see that, consider any so-called Fermat's prime, which is a prime number $p \in \mathbb{P}$, such that $p = 2^n + 1$ for some number n.

We know from exercise 31 that in this case $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ is a group with respect to integer multiplication in modular p arithmetics and since $p = 2^n + 1$, the order of $\mathbb{Z}_p^*$ is 2^n, which implies that the associated security parameter is given by $log_2(2^n) = n$.

check reference

We show that, in this case, $\mathbb{Z}_p^*$ is not a DL-A group, by constructing an algorithm, which is able compute some $x \in \mathbb{Z}_{2^n}$ for any given generator g and arbitrary element h of $\mathbb{F}_p^*$, such that equation 4.6 holds, and the runtime complexity of the constructed algorithm is $\mathcal{O}(n^2)$, which is quadratic in the security parameter $n = log_2(2^n)$.

$$h = g^x \tag{4.6}$$

To define such an algorithm, let us assume that the generator g is a public constant and that a group element h is given. Our task is to compute x efficiently.

The first thing to note is that, since x is a number in modular 2^n arithmetic, we can write the binary representation of x as in 4.7, with binary coefficients $c_j \in \{0, 1\}$. In particular, x is an n-bit number if interpreted as an integer.

explain last sentence more

$$x = c_0 \cdot 2^0 + c_1 \cdot 2^1 + \cdots + c_n \cdot 2^n \tag{4.7}$$

We then use this representation to construct an algorithm that computes the bits c_j one after another, starting at c_0. To see how this can be achieved, observe that we can determine c_0 by raising the input h to the power of 2^{n-1} in $\mathbb{F}_p^*$. We use the exponential laws and compute as follows:

$$\begin{aligned}
h^{2^{n-1}} &= (g^x)^{2^{n-1}} \\
&= \left(g^{c_0 \cdot 2^0 + c_1 \cdot 2^1 + \ldots + c_n \cdot 2^n}\right)^{2^{n-1}} \\
&= g^{c_0 \cdot 2^{n-1}} \cdot g^{c_1 \cdot 2^1 \cdot 2^{n-1}} \cdot g^{c_2 \cdot 2^2 \cdot 2^{n-1}} \cdots g^{c_n \cdot 2^n \cdot 2^{n-1}} \\
&= g^{c_0 2^{n-1}} \cdot g^{c_1 \cdot 2^0 \cdot 2^n} \cdot g^{c_2 \cdot 2^1 \cdot 2^n} \cdots g^{c_n \cdot 2^{n-1} \cdot 2^n}
\end{aligned}$$

Now, since g is a generator and $\mathbb{F}_p^*$ is cyclic of order 2^n, we know $g^{2^n} = 1$ and therefore $g^{k \cdot 2^n} = 1^k = 1$. From this, it follows that all but the first factor in the last expression are equal to 1 and we can simplify the expression into the following:

$$h^{2^{n-1}} = g^{c_0 2^{n-1}} \tag{4.8}$$

Now, in case $c_0 = 0$, we get $h^{2^{n-1}} = g^0 = 1$. In case $c_0 = 1$, we get $h^{2^{n-1}} = g^{2^{n-1}} \neq 1$ (To see that $g^{2^{n-1}} \neq 1$, recall that g is a generator of $\mathbb{F}_p^*$ and hence, is $\mathbb{F}_p^*$ a cyclic group of order 2^n, which implies $g^y \neq 1$ for all $y < 2^n$).

Raising h to the power of 2^{n-1} determines c_0, and we can apply the same reasoning to the coefficient c_1 by raising $h \cdot g^{-c_0 \cdot 2^0}$ to the power of 2^{n-2}. This approach can then be repeated until all the coefficients c_j of x are found.

Assuming that exponentiation in $\mathbb{F}_p^*$ can be done in logarithmic runtime complexity $log(p)$, it follows that our algorithm has a runtime complexity of $\mathcal{O}(log^2(p)) = \mathcal{O}(n^2)$, since we have to execute n exponentiations to determine the n binary coefficients of x.

From this, it follows that whenever p is a Fermat's prime, the discrete logarithm assumption does not hold in F_p^*.

The decisional Diffie–Hellman assumption Let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. The decisional Diffie–Hellman assumption stipulates that there is no algorithm that has a polynomial runtime complexity in the security parameter $s = log(r)$ that is able to distinguish the so-called DDH- triple (g^a, g^b, g^{ab}) from any triple (g^a, g^b, g^c) for randomly and independently chosen parameters $a, b, c \in \mathbb{Z}_r$. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a **DDH-A group**.

DDH-A is a stronger assumption than DL-A, in the sense that the discrete logarithm assumption is necessary for the decisional Diffie–Hellman assumption to hold, but not the other way around.

To see why this is the case, assume that the discrete logarithm assumption does not hold. In that case, given a generator g and a group element h, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. Then the decisional Diffie–Hellman assumption cannot hold, since given some triple (g^a, g^b, z), one could efficiently decide whether $z = g^{ab}$ is true by first computing the discrete logarithm b of g^b, then computing $g^{ab} = (g^a)^b$ and deciding whether or not $z = g^{ab}$.

On the other hand, the following example shows that there are groups where the discrete logarithm assumption holds but the decisional Diffie–Hellman assumption does not.

Example 41 (Efficiently computable pairings). Let $\mathbb{G}$ be a finite, cyclic group of order r with generator g, such that the discrete logarithm assumtion holds and there is a pairing map $e(\cdot,\cdot) : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$ that is computable in polynomial time of the parameter $log(r)$.

In a setting like this, it is easy to show that DDH-A cannot hold, since given some triple (g^a, g^b, z), it is possible to decide in polynomial time w.r.t $log(r)$ whether $z = g^{ab}$ or not. To see that, check the following equation:

$$e(g^a, g^b) = e(g, z) \tag{4.9}$$

Since the bilinearity properties of $e(\cdot,\cdot)$ imply $e(g^a, g^b) = e(g, g)^{ab} = e(g, g^{ab})$, and $e(g, y) = e(g, y')$ implies $y = y'$ due to the non-degenerate property, the equality means $z = e^{ab}$.

"equation"?

It follows that DDH-A is indeed weaker than DL-A, and groups with efficient pairings cannot be DDH-A groups. The following example shows another important class of groups where DDH-A does not hold: multiplicative groups of prime number residue classes.

Example 42. Let p be a prime number and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ the multiplicative group of modular p arithmetics as in exercise 31. As we have seen in XXX, this group is finite and cyclic of order $p-1$ and every element $g \neq 1$ is a generator.

check reference

To see that $\mathbb{F}_p^*$ cannot be a DDH-A group, recall from XXX that the Legendre symbol $\left(\frac{x}{p}\right)$ of any $x \in \mathbb{F}_p^*$ is efficiently computable by Euler's formular. But the Legendre symbol of g^a reveals whether a is even or odd. Given g^a, g^b and g^{ab}, one can thus efficiently compute and compare the least significant bit of a, b and ab, respectively, which provides a probabilistic method to distinguish g^{ab} from a random group element g^c.

what's the difference between $\mathbb{F}_p^*$ and $\mathbb{Z}_p^*$?

Legendre symbol

Euler's formular

These are only explained later in the text, '4.27'

The computational Diffie–Hellman assumption Let $\mathbb{G}$ be a finite cyclic group of order r and let g be a generator of $\mathbb{G}$. The computational Diffie–Hellman assumption stipulates that, given randomly and independently chosen residue classes $a, b \in \mathbb{Z}_r$, it is not possible to compute g^{ab} if only g, g^a and g^b (but not a and b) are known. If this is the case for $\mathbb{G}$, we call $\mathbb{G}$ a CDH-A group.

In general, we don't know if CDH-A is a stronger assumption then DL-A, or if both assumptions are equivalent. We know that DL-A is necessary for CDH-A, but the other direction is currently not well understood. In particular, there are no groups known where DL-A holds but CDH-A does not hold [**?**].

To see why the discrete logarithm assumption is necessary, assume that it does not hold. So, given a generator g and a group element h, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. In that case, the computational Diffie–Hellman assumption cannot hold, since, given g, g^a and g^b, one can efficiently compute b and hence is able to compute $g^{ab} = (g^a)^b$ from this data.

The computational Diffie–Hellman assumption is a weaker assumption than the decisional Diffie–Hellman assumption, which means that there are groups where CDH-A holds and DDH-A does not hold, while there cannot be groups such that DDH-A holds but CDH-A does not hold. To see that, assume that it is efficiently possible to compute g^{ab} from g, g^a and g^b. Then, given (g^a, g^b, z) it is easy to decide if $z = g^{ab}$ holds or not.

Several variations and special cases of the CDH-A exist. For example, the **square computational Diffie–Hellman assumption** assumes that, given g and g^x, it is computationally hard to compute g^{x^2}. The **inverse computational Diffie–Hellman assumption** assumes that, given g and g^x, it is computationally hard to compute $g^{x^{-1}}$.

are these going to be relevant later? yes, they are used in various snark proof systems

Cofactor Clearing

TODO: theorem: every factor of order defines a subgroup...

4.1.2 Hashing to Groups

Hash functions Generally speaking, a hash function is any function that can be used to map data of arbitrary size to fixed-size values. Since binary strings of arbitrary length are a general way to represent arbitrary data, we can understand a general **hash function** as the following map where $\{0,1\}^*$ represents the set of all binary strings of arbitrary but finite length and $\{0,1\}^k$ represents the set of all binary strings that have a length of exactly k bits:

$$H : \{0,1\}^* \to \{0,1\}^k \tag{4.10}$$

In our definition, a hash function maps binary strings of arbitrary size onto binary strings of size exactly k. The **images** of H, that is, the values returned by the hash function H, are called **hash values**, **digests**, or simply **hashes**.

A hash function must be deterministic, that is, when we insert the same input x into H, the image $H(x)$ must always be the same. In addition, a hash function should be as uniform as possible, which means that it should map input values as evenly as possible over its output range. In mathematical terms, every string of length k from $\{0,1\}^k$ should be generated with roughly the same probability.

Example 43 (k-truncation hash). One of the most basic hash functions $H_k : \{0,1\}^* \to \{0,1\}^k$ is given by simply truncating every binary string s of size $s.len() > k$ to a string of size k and by filling any string s' of size $s'.len() < 0$ with zeros. To make this hash function deterministic, we define that both truncation and filling should happen "on the left".

For example, if $k = 3$, $x_1 = (0000101011101010011101010101)$ and $x_2 = 1$, then $H(x_1) = (101)$ and $H(x_2) = (001)$. It is easy to see that this hash function is deterministic and uniform.

Of particular interest are so-called **cryptographic** hash functions, which are hash functions that are also **one-way functions**, which essentially means that, given a string y from $\{0,1\}^k$ it is practically infeasible to find a string $x \in \{0,1\}^*$ such that $H(x) = y$ holds. This property is usually called **preimage-resistance**.

In addition, it should be infeasible to find two strings $x_1, x_2 \in \{0,1\}^*$, such that $H(x_1) = H(x_2)$, which is called **collision resistance**. It is important to note, though, that collisions always exist, since a function $H : \{0,1\}^* \to \{0,1\}^k$ inevitably maps infinitely many values onto the same hash. In fact, for any hash function with digests of length k, finding a preimage to a given digest can always be done using a brute force search in 2^k evaluation steps. It should just be practically impossible to compute those values, and statistically very unlikely to generate two of them by chance.

A third property of a cryptographic hash function is that small changes in the input string, like changing a single bit, should generate hash values that look completely different from each other.

Is there a term for this property?

Because cryptographically secure hash functions map tiny changes in input values onto large changes in the output, implementation errors that change the outcome are usually easy to spot by comparing them to expected output values. The definitions of cryptographically secure hash functions are therefore usually accompanied by some test vectors of common inputs and expected digests. Since the empty string $''$ is the only string of length 0, a common test vector is the expected digest of the empty string.

Example 44 (k-truncation hash). Consider the k-truncation hash from example 43. Since the empty string has length 0, it follows that the digest of the empty string is string of length k that only contains 0's:

$$H_k('') = (000\ldots000) \tag{4.11}$$

It is pretty obvious from the definition of H_k that this simple hash function is not a cryptographic hash function. In particular, every digest is its own preimage, since $H_k(y) = y$ for every string of size exactly k. Finding preimages is therefore easy, so the property of preimage resistance does not hold.

In addition, it is easy to construct collisions as all strings of size $> k$ that share the same k-bits"on the right" are mapped to the same hash value, so this function is not collision resistant, either.

Finally, this hash function is not very chaotic, as changing bits that are not part of the k right-most bits don't change the digest at all.

Computing cryptographically secure hash functions in pen-and-paper style is possible but tedious. Fortunately, Sage can import the **hashlib** library, which is intended to provide a reliable and stable base for writing Python programs that require cryptographic functions. The following examples explain how to use hashlib in Sage.

Example 45. An example of a hash function that is generally believed to be a cryptographically secure hash function is the so-called **SHA256** hash, which, in our notation, is a function that maps binary strings of arbitrary length onto binary strings of length 256:

$$SHA256 : \{0,1\}^* \to \{0,1\}^{256} \tag{4.12}$$

To evaluate a proper implementation of the $SHA256$ hash function, the digest of the empty string is supposed to be

$$SHA256('') = e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 \tag{4.13}$$

For better human readability, it is common practice to represent the digest of a string not in its binary form, but in a hexadecimal representation. We can use Sage to compute $SHA256$ and freely transit between binary, hexadecimal and decimal representations. To do so, we import hashlib's implementation of SHA256:

```
sage: import hashlib
sage: test = 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934
    ca495991b7852b855'
sage: hasher = hashlib.sha256(b'')
sage: str = hasher.hexdigest()
sage: type(str)
<class 'str'>
sage: d = ZZ('0x'+ str) # conversion to integer type
sage: d.str(16) == str
True
sage: d.str(16) == test
True
sage: d.str(16)
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b8
    55
sage: d.str(2)
11100011101100001100010001000010100110001111110000011100000101
    00100110101111101111110100110010001001100101101111101110010
    01001000010011110101110010000011110010001100100100110111001
    00110100110010100100100101011001100100011011011110000101001
    01011100001010101
sage: d.str(10)
10298733624955409702953521232258132278979990064819803499337939
    7001115665086549
```

Hashing to cyclic groups As we have seen in the previous paragraph, general hash functions map binary strings of arbitrary length onto binary strings of length k. However, it is desirable in various cryptographic primitives to not simply hash to binary strings of fixed length but to hash into algebraic structures like groups, while keeping (some of) the properties like preimage resistance or collision resistance.

Hash functions like this can be defined for various algebraic structures, but, in a sense, the most fundamental ones are hash functions that map into groups, because they can be easily extended to map into other structures like rings or fields.

To give a more precise definition, let $\mathbb{G}$ be a group and $\{0,1\}^*$ the set of all finite, binary strings, then a **hash-to-group** function is a deterministic map

$$H : \{0,1\}^* \to \mathbb{G} \tag{4.14}$$

Common properties of hash functions, like uniformity, are desirable but not always realized in real-world instantiations of hash-to-group functions, so we skip those requirements for now and keep the definition very general.

As the following example shows, hashing to finite cyclic groups can be trivially achieved for the price of some undesirable properties of the hash function:

Example 46 (Naive cyclic group hash). Let $\mathbb{G}$ be a finite cyclic group. If the task is to implement a hash-to-group function, one immediate approach can be based on the observation that binary strings of size k can be interpreted as integers $z \in \mathbb{Z}$ in the range $0 \leq z < 2^k$.

To be more precise, choose an ordinary hash function $H : \{0,1\}^* \to \{0,1\}^k$ for some parameter k and a generator g of $\mathbb{G}$. Then the expression below is a positive integer (where $H(s)_j$ means the bit at the j-th position of $H(s)$):

$$z_{H(s)} = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_k \cdot 2^k \tag{4.15}$$

A hash-to-group function for the group $\mathbb{G}$ can then be defined as a concatenation of the exponential map $g^{(\cdot)}$ of g with the interpretation of $H(s)$ as an integer:

$$H_g : \{0,1\}^* \to \mathbb{G} : \; s \mapsto g^{z_{H(s)}} \tag{4.16}$$

Constructing a hash-to-group function like this is easy for cyclic groups, and it might be good enough in certain applications. It is, however, almost never adequate in cryptographic applications, as discrete log relations might be constructible between two given hash values $H_g(s)$ and $H_g(t)$. a few examples?

To see that, assume that $\mathbb{G}$ is of order r and that $z_{H(s)}$ has a multiplicative inverse in modular r arithmetics. In that case, we can compute $x = z_{H(t)} \cdot z_{H(s)}^{-1}$ in $\mathbb{Z}_r$ and find a discrete log relation between the group hash values, that is, find some x with $H_g(t) = (H_g(s))^x$:

$$\begin{aligned}
H_g(t) &= (H_g(s))^x && \Leftrightarrow\\
g^{z_{H(t)}} &= g^{z_{H(s)} \cdot x} && \Leftrightarrow\\
g^{z_{H(t)}} &= g^{z_{H(t)}}
\end{aligned}$$

Therefore applications where discrete log relations between hash values are undesirable need different approaches. Many of these approaches start with a way to hash into the set $\mathbb{Z}_r$ of modular r arithmetics.

Hashing to modular arithmetics One of the most widely used applications of hash-into-group functions are hash functions that map into the set $\mathbb{Z}_r$ of modular r arithmetics for some modulus r. Different approaches to construct such a function are known, but probably the most widely used ones are based on the insight that the images of arbitrary hash functions can be interpreted as binary representations of integers, as explained in example 46. check reference

It follows from this interpretation that one simple method of hashing into $\mathbb{Z}_r$ is constructed by observing that if r is a modulus with a bit-length of $k = r.nbits()$, then every binary string $(b_0, b_1, \ldots, b_{k-2})$ of length $k-1$ defines an integer z in the rage $0 \leq z < 2^{k-1} \leq r$, by defining z:

$$z = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_{k-2} \cdot 2^{k-2} \tag{4.17}$$

Now, since $z < r$, we know that z is guaranteed to be in the set $\{0, 1, \ldots, r-1\}$, and hence it can be interpreted as an element of $\mathbb{Z}_r$. From this it follows that if $H : \{0,1\}^* \to \{0,1\}^{k-1}$ is a hash function, then a hash-to-group function can be constructed as follows (where $H(s)_j$ means the j-th bit of the image binary string $H(s)$ of the original binary hash function):

$$H_{r.nbits()-1} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-2} \cdot 2^{k-2} \tag{4.18}$$

A drawback of this hash function is that the distribution of the hash values in $\mathbb{Z}_r$ is not necessarily uniform. In fact, if $r - 2^{k-1} \neq 0$, then by design $H_{r.nbits()-1}$ will never hash onto values $z \geq 2^{k-1}$. Good moduli r are therefore as close to 2^{k-1} as possible, while less good moduli are closer to 2^k. In the worst case, when $r = 2^k - 1$, it misses $2^{k-1} - 1$, that is, almost half of all elements, from $\mathbb{Z}_r$. TODO: DOUBLE CHECK THIS REASONING.

An advantage of this approach is that properties like preimage resistance or collision resistance of the original hash function $H(\cdot)$ are preserved.

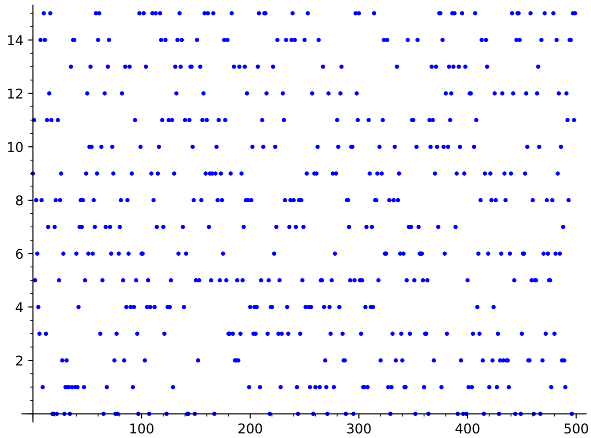
Example 47. To give an implementation of the $H_{r.nbits()-1}$ hash function, we use a 5-bit truncation of the *SHA*256 hash from example 45 and define a hash into $\mathbb{Z}_{16}$ as follows:

$$H_{16.nbits()-1} : \{0,1\}^* \to \mathbb{Z}_{16} : s \mapsto SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_4 \cdot 2^4$$

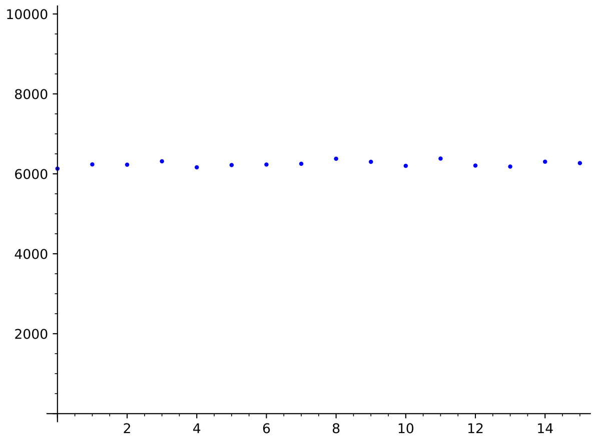
Since $k = 16.nbits() = 5$ and $16 - 2^{k-1} = 0$, this hash maps uniformly onto $\mathbb{Z}_{16}$. We can invoke Sage to implement it:

```
sage: import hashlib
sage: def Hash5(x):
....:     hasher = hashlib.sha256(x)
....:     digest = hasher.hexdigest()
....:     d = ZZ(digest, base=16)
....:     d = d.str(2)[-4:]
....:     return ZZ(d,base=2)
sage: Hash5(b'')
5
```

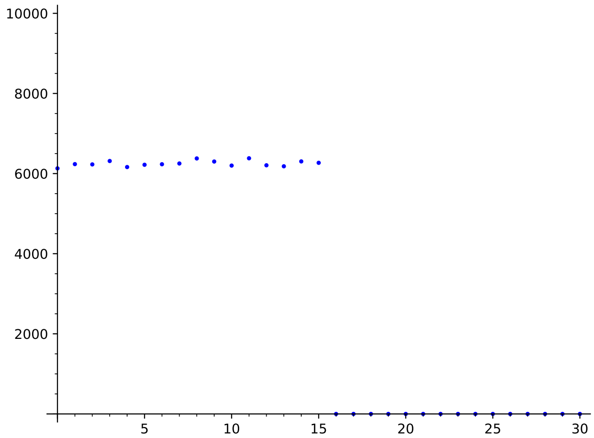
We can then use Sage to apply this function to a large set of input values in order to plot a visualization of the distribution over the set $\{0,\ldots,15\}$. Executing over 500 input values gives the following plot:



To get an intuition of uniformity, we can count the number of times the hash function $H_{16.nbits()-1}$ maps onto each number in the set $\{0,1,\ldots,15\}$ in a loop of 100000 hashes, and compare that to the ideal uniform distribution, which would map exactly 6250 samples to each element. This gives the following result:



The uniformity of distribution problem becomes apparent if we want to construct a similar hash function for $\mathbb{Z}_r$ for any r in the range $17 \leq r \leq 31$. In this case, the definition of the hash function is exactly the same as for $\mathbb{Z}_{16}$, and hence, the images will not exceed the value 16. So, for example, even in the case of hashing to $\mathbb{Z}_{31}$, the hash function never maps to any value larger than 16, leaving almost half of all numbers out of the image range.



Mirco: We can do better than this

The second widely used method of hashing into $\mathbb{Z}_r$ is constructed by observing the following: If r is a modulus with a bit-length of $r.bits() = k_1$ and $H : \{0,1\}^* \to \{0,1\}^{k_2}$ is a hash function that produces digests of size k_2, with $k_2 \geq k_1$, then a hash-to-group function can be constructed by interpreting the image of H as a binary representation of an integer and then taking the modulus by r to map into $\mathbb{Z}_r$. This is formalized in the equation below, where $H(s)_j$ means the j'th bit of the image binary string $H(s)$ of the original binary hash function.

$$H'_{mod_r} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto \left(H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k_2} \cdot 2^{k_2} \right) \bmod n \qquad (4.19)$$

A drawback of this hash function is that computing the modulus requires some computational effort. In addition, the distribution of the hash values in $\mathbb{Z}_r$ might not be even, depending on the difference $2^{k_2+1} - r$. An advantage of it is that potential properties of the original hash function $H(\cdot)$(like preimage resistance or collision resistance) are preserved, and the distribution can be made almost uniform, with only negligible bias depending on what modulus r and images size k_2 are chosen.

Example 48. To give an implementation of the H_{mod_r} hash function, we use k_2-bit truncation of the $SHA256$ hash from example 45, and define a hash into $\mathbb{Z}_{23}$ as follows:

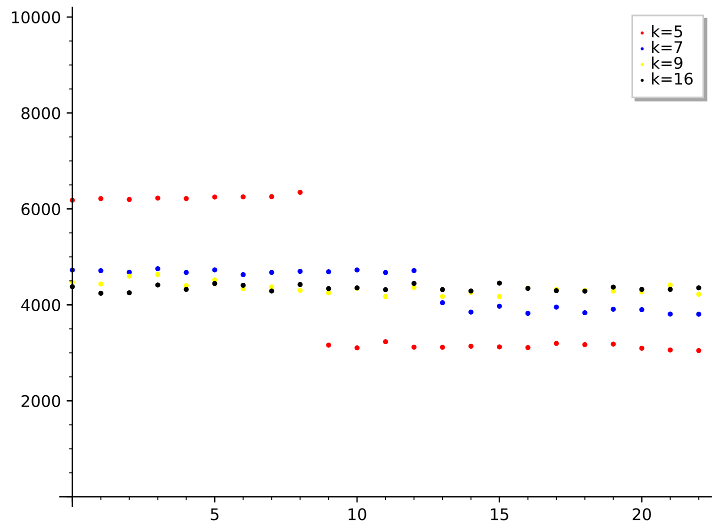
$$\begin{aligned} &H_{mod_{23},k_2} : \{0,1\}^* \to \mathbb{Z}_{23} : \\ &\qquad s \mapsto \left(SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_{k_2} \cdot 2^{k_2} \right) \bmod 23 \end{aligned}$$

We want to use various instantiations of k_2 to visualize the impact of truncation length on the distribution of the hashes in $\mathbb{Z}_{23}$. We can invoke Sage to implement it as follows:

```
sage: import hashlib
sage: Z23 = Integers(23)
sage: def Hash_mod23(x, k2):
....:     hasher = hashlib.sha256(x.encode('utf-8'))
....:     digest = hasher.hexdigest()
....:     d = ZZ(digest, base=16)
```

```
....:       d = d.str(2)[-k2:]
....:       d = ZZ(d, base=2)
....:       return Z23(d)
```

We can then use Sage to apply this function to a large set of input values in order to plot visualizations of the distribution over the set $\{0,\ldots,22\}$ for various values of k_2, by counting the number of times it maps onto each number in a loop of 100000 hashes. We get the following plot:



A third method that can sometimes be found in implementations is the so-called **"try-and-increment" method**. To understand this method, we define an integer $z \in \mathbb{Z}$ from any hash value $H(s)$ as we did in the previous methods, that is, we define $z = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-1} \cdot 2^k$.

Hashing into $\mathbb{Z}_r$ is then achievable by first computing z, and then trying to see if $z \in \mathbb{Z}_r$. If it is, then the hash is done; if not, the string s is modified in a deterministic way and the process is repeated until a suitable number z is found. A suitable, deterministic modification could be to concatenate the original string by some bit counter. A "try-and-increment" algorithm would then work like in algorithm 5.

check reference

Algorithm 5 Hash-to-$\mathbb{Z}_n$

Require: $r \in \mathbb{Z}$ with $r.nbits() = k$ and $s \in \{0,1\}^*$
  procedure TRY-AND-INCREMENT(r,k,s)
    $c \leftarrow 0$
    repeat
      $s' \leftarrow s||c\_bits()$
      $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \ldots + H(s')_k \cdot 2^k$
      $c \leftarrow c+1$
    until $z < r$
    return x
  end procedure
Ensure: $z \in \mathbb{Z}_r$

Depending on the parameters, this method can be very efficient. In fact, if k is sufficiently large and r is close to 2^{k+1}, the probability for $z < r$ is very high and the repeat loop will almost always be executed a single time only. A drawback is, however, that the probability of having to execute the loop multiple times is not zero.

Once some hash function into modular arithmetics is found, it can often be combined with additional techniques to hash into more general finite cyclic groups. The following paragraphs describe a few of those methods widely adopted in SNARK development.

Pedersen Hashes The so-called **Pedersen hash function** [?] provides a way to map binary inputs of fixed size k onto elements of finite cyclic groups that avoids discrete log relations between the images as they occur in the naive approach XXX. Combining it with a classical hash function provides a hash function that maps strings of arbitrary length onto group elements. add reference

To be more precise, let j be an integer, $\mathbb{G}$ a finite cyclic group of order r and $\{g_1,\ldots,g_j\} \subset \mathbb{G}$ a uniform randomly generated set of generators of $\mathbb{G}$. Then **Pedersen's hash function** is defined as follows:

$$H_{Ped} : (\mathbb{Z}_r)^j \to \mathbb{G} : \; (x_1,\ldots,x_j) \mapsto \Pi_{i=1}^{j} g_j^{x_j} \tag{4.20}$$

It can be shown that Pedersen's hash function is collision-resistant under the assumption that $\mathbb{G}$ is a DL-A group. However, it is important to note that Pedersen hashes cannot be assumed to be pseudorandom and should therefore not be used where a hash function serves as an approximation of a random oracle.will these be explained in the initial chapters? pseudorandom oracle

From an implementation perspective, it is important to derive the set of generators $\{g_1,\ldots,g_j\}$ in such a way that they are as uniform and random as possible. In particular, any known discrete log relation between two generators, that is, any known $x \in \mathbb{Z}_r$ with $g_h = (g_i)^x$ must be avoided.

To see how Pedersen hashes can be used to define an actual hash-to-group function according to our definition, we can use any of the hash-to-$\mathbb{Z}_r$ functions as we have derived them in equation 4.18. check reference

MimC Hashes [?] add text on this

Pseudorandom Functions in DDH-A groups As noted in above, Pederson's hash function does not have the properties a random function and should therefore not be instantiated as such. To see an example of a random oracle function in groups where the decisional Diffie–Hellman construction is assumed to hold true, let $\mathbb{G}$ be a DDH-A group of order r with generator g and $\{a_0,a_1,\ldots,a_k\} \subset \mathbb{Z}_r^*$ a uniform randomly generated set of numbers invertible in modular r arithmetics. Then a pseudo-random function is given by the as follows:

$$F_{rand} : \{0,1\}^{k+1} \to \mathbb{G} : \; (b_0,\ldots,b_k) \mapsto g^{b_0 \cdot \Pi_{i=1}^{k} a_i^{b_i}} \tag{4.21}$$

Of course, if $H : \{0,1\}^* \to \{0,1\}^{k+1}$ is a random oracle, then the concatenation of F_{rand} and H also defines a random oracle

$$H_{rand,\mathbb{G}} : \{0,1\}^* \to \mathbb{G} : \; s \mapsto F_{rand}(H(s)) \tag{4.22}$$

4.2 Commutative Rings

Thinking back to operations on integers, we know that there are two of these: addition and multiplication. As we have seen, addition defines a group structure on the set of integers. However, multiplication does not define a group structure, given that integers generally don't have multiplicative inverses.

Configurations like these constitute so-called **commutative rings with unit**. To be more precise, a commutative ring with unit $(R,+,\cdot,1)$ is a set R provided with two maps $+ : R\cdot R \to R$ and $\cdot : R\cdot R \to R$, called **addition** and **multiplication**, such that the following conditions hold:

Definition 4.2.0.1. **Commutative ring with unit**

- $(R,+)$ is a commutative group, where the neutral element is denoted with 0. **Commutativity of multiplication**: $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.
- **Existence of a unit**: There is an element $1 \in R$, such that $1 \cdot g$ holds for all $g \in R$,
- **Associativity**: For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.
- **Distributivity**: For all $g_1, g_2, g_3 \in R$ the distributive laws $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

Example 49 (The ring of integers). The set $\mathbb{Z}$ of integers with the usual addition and multiplication is the archetypical example of a commutative ring with unit 1.

Example 50 (Underlying commutative group of a ring). Every commutative ring with unit $(R,+,\cdot,1)$ gives rise to a group, if we disregard multiplication.

The following example is somewhat unusual, but we encourage you to think through it because it helps to detach the mind from familiar styles of computation and concentrate on the abstract algebraic explanation.

Example 51. Let $S := \{\bullet, \star, \odot, \otimes\}$ be a set that contains four elements, and let addition and multiplication on S be defined as follows:

$\cup$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\star$	$\odot$	$\otimes$
$\star$	$\star$	$\odot$	$\otimes$	$\bullet$
$\odot$	$\odot$	$\otimes$	$\bullet$	$\star$
$\otimes$	$\otimes$	$\bullet$	$\star$	$\odot$

$\circ$	$\bullet$	$\star$	$\odot$	$\otimes$
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\star$	$\bullet$	$\star$	$\odot$	$\otimes$
$\odot$	$\bullet$	$\odot$	$\bullet$	$\odot$
$\otimes$	$\bullet$	$\otimes$	$\odot$	$\star$

Then $(S,\cup,\circ)$ is a ring with unit $\star$ and zero $\bullet$. It therefore makes sense to ask for solutions to equations like this one: Find $x \in S$ such that

$$\otimes \circ (x \cup \odot) = \star$$

To see how such a "moonmath equation" can be solved, we have to keep in mind that rings behaves mostly like normal numbers when it comes to bracketing and computation rules. The only differences are the symbols, and the actual way to add and multiply them. With this in mind, we solve the equation for x in the“usual way”:

$$
\begin{aligned}
\otimes \circ (x \cup \odot) &= \star && \text{\# apply the distributive law}\\
\otimes \circ x \cup \otimes \circ \odot &= \star && \# \otimes \circ \odot = \odot\\
\otimes \circ x \cup \odot &= \star && \text{\# concatenate the } \cup \text{ inverse of } \odot \text{ to both sides}\\
\otimes \circ x \cup \odot \cup -\odot &= \star \cup -\odot && \# \odot \cup -\odot = \bullet\\
\otimes \circ x \cup \bullet &= \star \cup -\odot && \# \bullet \text{ is the } \cup \text{ neutral element}\\
\otimes \circ x &= \star \cup -\odot && \text{\# for } \cup \text{ we have } -\odot = \odot\\
\otimes \circ x &= \star \cup \odot && \# \star \cup \odot = \otimes\\
\otimes \circ x &= \otimes && \text{\# concatenate the } \circ \text{ inverse of } \otimes \text{ to both sides}\\
(\otimes)^{-1} \circ \otimes \circ x &= (\otimes)^{-1} \circ \otimes && \text{\# multiply with the multiplicative inverse}\\
\star \circ x &= \star\\
x &= \star
\end{aligned}
$$

So, even though this equation looked really alien at first glance, we could solve it basically exactly the way we solve “normal” equations containing numbers.

Note, however, that whenever a multiplicative inverse would be needed to solve an equation in the usual way in a ring, things can be very different than most of us are used to. For example, the following equation cannot be solved for x in the usual way, since there is no multiplicative inverse for $\odot$ in our ring.

$$\odot \circ x = \otimes \tag{4.23}$$

We can confirm this by looking at the multiplication table to see that no such x exits.

As another example, the following equation does not have a single solution but two: $x \in \{\star, \otimes\}$.

$$\odot \circ x = \odot \tag{4.24}$$

Having no solution or two solutions is certainly not something to expect from types like $\mathbb{Q}$ (rational numbers). can we use another set as an example? we hardly talked about Q so far

Example 52. Considering polynomials again, we note from their definition that what we have called the type R of the coefficients must in fact be a commutative ring with a unit, since we need addition, multiplication, commutativity and the existence of a unit for $R[x]$ to have the properties we expect.

Considering R to be a ring with addition and multiplication of polynomials as defined in 4.2.0.1 actually makes $R[x]$ into a commutative ring with a unit, too, where the polynomial 1 is the multiplicative unit. check reference

Example 53. Let n be a modulus and $(\mathbb{Z}_n, +, \cdot)$ the set of all remainder classes of integers modulo n, with the projection of integer addition and multiplication as defined in 4.2.0.1. It can be shown that $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring with unit 1. check reference

Considering the exponential map from page 44 again, let $\mathbb{G}$ be a finite cyclic group of order n with generator $g \in \mathbb{G}$. Then the ring structure of $(\mathbb{Z}_n, +, \cdot)$ is mapped onto the group structure of $\mathbb{G}$ in the following way: check reference

$$
\begin{aligned}
g^{x+y} &= g^x \cdot g^y && \text{for all } x, y \in \mathbb{Z}_n\\
g^{x \cdot y} &= (g^x)^y && \text{for all } x, y \in \mathbb{Z}_n
\end{aligned}
$$

This is of particular interest in cryptography and SNARKs, as it allows for the evaluation of polynomials with coefficients in $\mathbb{Z}_n$ to be evaluated “in the exponent”. To be more precise, let $p \in \mathbb{Z}_n[x]$ be a polynomial with $p(x) = a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$. Then the previously defined exponential laws (example 37) imply the following:

check reference

$$
\begin{aligned}
g^{p(x)} &= g^{a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0} \\
&= \left(g^{x^m}\right)^{a_m} \cdot \left(g^{x^{m-1}}\right)^{a_{m-1}} \cdot \ldots \cdot (g^x)^{a_1} \cdot g^{a_0}
\end{aligned}
$$

Hence, to evaluate p at some point s in the exponent, we can insert s into the right-hand side of the last equation and evaluate the product.

As we will see, this is a key insight to understanding many SNARK protocols like e.g. Groth16 [?] or XXX.

add more examples protocols of SNARK

Example 54. To give an example of the evaluation of a polynomial in the exponent of a finite cyclic group, consider the exponential map from example 37:

check reference

$$
3^{(\cdot)} : \mathbb{Z}_4 \to \mathbb{Z}_5^* \; x \mapsto 3^x
$$

Choosing the polynomial $p(x) = 2x^2 + 3x + 1$ from $\mathbb{Z}_4[x]$, we can evaluate the polynomial at say $x = 2$ in the exponent of 3 in two different ways. On the one hand, we can evaluate p at 2 and then write the result into the exponent as follows:

$$
\begin{aligned}
3^{p(2)} &= 3^{2 \cdot 2^2 + 3 \cdot 2 + 1} \\
&= 3^{2 \cdot 0 + 2 + 1} \\
&= 3^3 \\
&= 2
\end{aligned}
$$

On the other hand, we can use the right-hand side of the equation to evaluate p at 2 in the exponent of 3 as follows:

$$
\begin{aligned}
3^{p(2)} &= \left(3^{2^2}\right)^2 \cdot \left(3^2\right)^3 \cdot 3^1 \\
&= \left(3^0\right)^2 \cdot 3^3 \cdot 3 \\
&= 1^2 \cdot 2 \cdot 3 \\
&= 2 \cdot 3 \\
&= 2
\end{aligned}
$$

Hashing to Commutative Rings As we have seen in XXX, various constructions for hashing-to-groups are known and used in applications. As commutative rings are Abelian groups, when we disregard the multiplicative structure, hash-to-group constructions can be applied for hashing into commutative rings, too. This is possible in general, as the codomain of a general hash function $\{0,1\}^*$ is just the set of binary strings of arbitrary but finite length, which has no algebraic structure that the hash function must respect.

add reference

Abelian groups

codomain

4.3 Fields

We started this chapter with the definition of a group (section 4.1), which we then expanded into the definition of a commutative ring with a unit (section 4.2). Such rings generalize the behavior of integers. In this section, we will look at those special cases of commutative rings where every element other than the neutral element of addition has a multiplicative inverse. Those structures behave very much like the rational numbers $\mathbb{Q}$. Rational numbers are, in a sense, an extension of the ring of integers, that is, they are constructed by including newly defined multiplicative inverses (fractions) to the integers.

Now, considering the definition of a ring (4.2.0.1) again, we define a **field** $(\mathbb{F},+,\cdot)$ to be a set $\mathbb{F}$, together with two maps $+ : \mathbb{F}\cdot\mathbb{F} \to \mathbb{F}$ and $\cdot : \mathbb{F}\cdot\mathbb{F} \to \mathbb{F}$, called *addition* and *multiplication*, such that the following conditions hold:

Definition 4.3.0.1. **Field**

- $(\mathbb{F},+)$ is a commutative group, where the neutral element is denoted by 0.
- $(\mathbb{F}\setminus\{0\},\cdot)$ is a commutative group, where the neutral element is denoted by 1.
- (Distributivity) For all $g_1,g_2,g_3 \in \mathbb{F}$ the distributive law $g_1\cdot(g_2+g_3) = g_1\cdot g_2 + g_1\cdot g_3$ holds.

If a field is given and the definition of its addition and multiplication is not ambiguous, we will often simply write $\mathbb{F}$ instead of $(\mathbb{F},+,\cdot)$ to denote the field. Moreover, we use $\mathbb{F}^*$ to describe the multiplicative group of the field, that is, the set of elements with multiplication as the group law, excluding the neutral element of addition.

The **characteristic** of a field $\mathbb{F}$, represented as $char(\mathbb{F})$, is the smallest natural number $n \geq 1$ for which the n-fold sum of 1 equals zero, i.e. for which $\sum_{i=1}^{n} 1 = 0$. If such an $n > 0$ exists, the field is also said to have a **finite characteristic**. If, on the other hand, every finite sum of 1 is such that it is not equal to zero, then the field is defined to have characteristic 0. S: Tried to disambiguate the scope of negation between 1. "It is true of every finite sum of 1 that it is not equal to 0" and 2. "It is not true of every finite sum of 1 that it is equal to 0" From the example below, it looks like 1. is the intended meaning here, correct?

Check change of wording

Example 55 (Field of rational numbers). Probably the best known example of a field is the set of rational numbers $\mathbb{Q}$ together with the usual definition of addition, subtraction, multiplication and division. Since there is no natural number $n \in \mathbb{N}$, such that $\sum_{j=0}^{n} 1 = 0$ in the set of rational numbers, the characteristic $char(\mathbb{Q})$ of the field $\mathbb{Q}$ is zero. In Sage, rational numbers are called as follows:

```
sage: QQ
Rational Field
sage: QQ(1/5) # Get an element from the field of rational
    numbers
1/5
sage: QQ(1/5) / QQ(3) # Division
1/15
```

Example 56 (Field with two elements). It can be shown that, in any field, the neutral element 0 of addition must be different from the neutral element 1 of multiplication, that is, $0 \neq 1$ always holds in a field. From this, it follows that the smallest field must contain at least two elements. As the following addition and multiplication tables show, there is indeed a field with two elements, which is usually called $\mathbb{F}_2$:

Let $\mathbb{F}_2 := \{0,1\}$ be a set that contains two elements and let addition and multiplication on $\mathbb{F}_2$ be defined as follows:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Since $1+1=0$ in the field $\mathbb{F}_2$, we know that the characteristic of $\mathbb{F}_2$ is there, that is, we have $char(\mathbb{F}_2)=0$.

For reasons we will understand better in XXX, Sage defines this field as a so-called Galois field with 2 elements. You can call it in Sage as follows:

add reference

```
sage: F2 = GF(2)
sage: F2(1) # Get an element from GF(2)
1
sage: F2(1) + F2(1) # Addition
0
sage: F2(1) / F2(1) # Division
1
```

Example 57. Both the real numbers $\mathbb{R}$ as well as the complex numbers $\mathbb{C}$ are well known examples of fields.

Expand on this?

Exercise 34. Consider our remainder class ring $(\mathbb{Z}_5,+,\cdot)$ and show that it is a field. What is the characteristic of $\mathbb{Z}_5$?

Prime fields As we have seen in the various examples of the previous sections, modular arithmetics behaves similarly to the ordinary arithmetics of integers in many ways. This is due to the fact that remainder class sets $\mathbb{Z}_n$ are commutative rings with units.

However, we have also seen in 36 that, whenever the modulus is a prime number, every remainder class other than the zero class has a modular multiplicative inverse. This is an important observation, since it immediately implies that, in case of a prime number, the remainder class set $\mathbb{Z}_n$ is not just a ring but actually a **field**. Moreover, since $\sum_{j=0}^{n} 1 = 0$ in $\mathbb{Z}_n$, we know that those fields have the finite characteristic n.

check reference

To distinguish this important case from arbitrary remainder class rings, we write $(\mathbb{F}_p,+,\cdot)$ for the field of all remainder classes for a prime number modulus $p\in\mathbb{P}$ and call it the **prime field** of characteristic p.

Prime fields are the foundation for many of the contemporary algebra-based cryptographic systems, as they have many desirable properties. One of them is that, since these sets are finite and a prime field of characteristic, p can be represented on a computer in roughly $log_2(p)$ amount of space without precision problems that are unavoidable for computer representations of infinite sets such as rational numbers or integers.

Since prime fields are special cases of remainder class rings, all computations remain the same. Addition and multiplication can be computed by first doing normal integer addition and multiplication, and then taking the remainder modulus p. Subtraction and division can be computed by adding or multiplying with the additive or the multiplicative inverse, respectively. The additive inverse $-x$ of a field element $x\in\mathbb{F}_p$ is given by $p-x$, and the multiplicative inverse of $x\neq 0$ is given by x^{p-2}, or can be computed using the Extended Euclidean Algorithm.

Note that these computations might not be the fastest to implement on a computer. They are, however, useful in this book, as they are easy to compute for small prime numbers.

Example 58. The smallest field is the field $\mathbb{F}_2$ of characteristic 2 as we have seen in example 56. It is the prime field of the prime number 2.

Example 59. To summarize the basic aspects of computation in prime fields, let us consider the prime field $\mathbb{F}_5$ and simplify the following expression:

$$\left(\frac{2}{3}-2\right)\cdot 2$$

A first thing to note is that since $\mathbb{F}_5$ is a field, all rules are identical to the rules we learned in school when we where dealing with rational, real or complex numbers. This means we can use e.g. bracketing (distributivity) or addition as usual:

$$\begin{aligned}
\left(\frac{2}{3}-2\right)\cdot 2 &= \frac{2}{3}\cdot 2 - 2\cdot 2 && \text{\# distributive law}\\
&= \frac{2\cdot 2}{3} - 2\cdot 2 && 4 \bmod 5 = 4\\
&= \frac{4}{3} - 4 && \text{\# multiplicative inverse of 3 is } 3^{5-2} \bmod 5 = 2\\
&= 4\cdot 2 - 4 && \text{\# additive inverse of 4 is } 5-4=1\\
&= 4\cdot 2 + 1 && 8 \bmod 5 = 3\\
&= 3+1 && 4 \bmod 5 = 4\\
&= 4
\end{aligned}$$

In this computation, we computed the multiplicative inverse of 3 using the identity $x^{-1} = x^{p-2}$ in a prime field. This is impractical for large prime numbers. Recall that another way of computing the multiplicative inverse is the Extended Euclidean Algorithm (see 3.10 on page 19). To refresh our memory, the task is to compute $x^{-1}\cdot 3 + t\cdot 5 = 1$, but t is actually irrelevant. We get

k	r_k	x_k^{-1}	$t_k = (r_k - s_k\cdot a)$ div b
0	3	1	·
1	5	0	·
2	3	1	·
3	2	-1	·
4	1	2	·

So the multiplicative inverse of 3 in $\mathbb{Z}_5$ is 2, and, indeed, if compute $3\cdot 2$, we get 1 in $\mathbb{F}_5$.

Square Roots In this part, we deal with square numbers, also called **quadratic residues** and **square roots** in prime fields. This is of particular importance in our studies on elliptic curves, as only square numbers can actually be points on an elliptic curve.

S: are we introducing elliptic curves in section 1 or 2?

To make the intuition of quadratic residues and roots precise, let $p \in \mathbb{P}$ be a prime number and $\mathbb{F}_p$ its associated prime field. Then a number $x \in \mathbb{F}_p$ is called a **square root** of another number $y \in \mathbb{F}_p$, if x is a solution to the following equation:

$$x^2 = y \tag{4.25}$$

In this case, y is called a **quadratic residue**. On the other hand, if y is given and the quadratic equation has no solution x , we call y a **quadratic non-residue**. For any $y \in \mathbb{F}_p$, we denote the set of all square roots of y in the prime field $\mathbb{F}_n$ as follows:

$$\sqrt{y} := \{x \in \mathbb{F}_p \mid x^2 = y\} \tag{4.26}$$

If y is a quadratic non-residue, then $\sqrt{y} = \emptyset$ (an empty set), and if $y = 0$, then $\sqrt{y} = \{0\}$.

Informally speaking, quadratic residues are numbers such that we can take the square root of them, while quadratic non-residues are numbers that don't have square roots. The situation therefore parallels the familiar case of integers, where some integers like 4 or 9 have square roots and others like 2 or 3 don't (as integers).

It can be shown that, in any prime field, every non zero element has either no square root or two of them. We adopt the convention to call the smaller one (when interpreted as an integer) as the **positive** square root and the larger one as the **negative**. This makes sense, as the larger one can always be computed as the modulus minus the smaller one, which is the definition of the negative in prime fields.

Example 60 (Quadratic (Non)-Residues and roots in $\mathbb{F}_5$). Let us consider our example prime field $\mathbb{F}_5$ again. All square numbers can be found on the main diagonal of the multiplication table in example 13 on page 28. As you can see, in $\mathbb{F}_5$ only the numbers 0, 1 and 4 have square roots and we get $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,4\}$, $\sqrt{2} = \emptyset$, $\sqrt{3} = \emptyset$ and $\sqrt{4} = \{2,3\}$. The numbers 0, 1 and 4 are therefore quadratic residues, while the numbers 2 and 3 are quadratic non-residues. check reference

In order to describe whether an element of a prime field is a square number or not, the so-called **Legendre symbol** can sometimes be found in the literature, which is why we will summerize it here:

Let $p \in \mathbb{P}$ be a prime number and $y \in \mathbb{F}_p$ an element from the associated prime field. Then the *Legendre symbol* of y is defined as follows:

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \text{if } y \text{ has square roots} \\ -1 & \text{if } y \text{ has no square roots} \\ 0 & \text{if } y = 0 \end{cases} \tag{4.27}$$

Example 61. Looking at the quadratic residues and non residues in $\mathbb{F}_5$ from example 13 again, we can deduce the following Legendre symbols, from example XXX. check reference add reference

$$\left(\tfrac{0}{5}\right) = 0, \quad \left(\tfrac{1}{5}\right) = 1, \quad \left(\tfrac{2}{5}\right) = -1, \quad \left(\tfrac{3}{5}\right) = -1, \quad \left(\tfrac{4}{5}\right) = 1\,.$$

The Legendre symbol provides a criterion to decide whether or not an element from a prime field has a quadratic root or not. This, however, is not just of theoretical use: The so-called **Euler criterion** provides a compact way to actually compute the Legendre symbol. To see that, let $p \in \mathbb{P}_{\geq 3}$ be an odd prime number and $y \in \mathbb{F}_p$. Then the Legendre symbol can be computed as follows:

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}}\,. \tag{4.28}$$

Example 62. Looking at the quadratic residues and non residues in $\mathbb{F}_5$ from example 13 again, check reference we can compute the following Legendre symbols using the Euler criterion:

$$\left(\frac{0}{5}\right) = 0^{\frac{5-1}{2}} = 0^2 = 0$$
$$\left(\frac{1}{5}\right) = 1^{\frac{5-1}{2}} = 1^2 = 1$$
$$\left(\frac{2}{5}\right) = 2^{\frac{5-1}{2}} = 2^2 = 4 = -1$$
$$\left(\frac{3}{5}\right) = 3^{\frac{5-1}{2}} = 3^2 = 4 = -1$$
$$\left(\frac{4}{5}\right) = 4^{\frac{5-1}{2}} = 4^2 = 1$$

Exercise 35. Consider the prime field $\mathbb{F}_{13}$. Find the set of all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ that satisfy the equation

$$x^2 + y^2 = 1 + 7 \cdot x^2 \cdot y^2$$

Exponentiation TO APPEAR...

write paragraph on exponentiation

Hashing into prime fields An important problem in SNARK development is the ability to hash to (various subsets) of elliptic curves. As we will see in XXX, those curves are often defined over prime fields, and hashing to a curve might start with hashing to the prime field. It is therefore important to understand how to hash into prime fields.

add reference

On pages 52–56, we looked at a few methods of hashing into the residue class rings $\mathbb{Z}_n$ for arbitrary $n > 1$. As prime fields are just special instances of those rings, all methods for hashing into $\mathbb{Z}_n$ functions can be used for hashing into prime fields, too.

check reference

Extension Fields Prime fields, defined in the previous section, are the basic building blocks for cryptography in general and SNARKs in particular.

However, as we will see in XXX so-called **pairing-based** SNARK systems are crucially dependent on group pairings XXX defined over the group of rational points of elliptic curves. For those pairings to be non-trivial, the elliptic curve must not only be defined over a prime field, but over a so-called **extension field** of a given prime field.

add reference

group pairings

We therefore have to understand field extensions. First note that the field $\mathbb{F}'$ is called an **extension** of a field $\mathbb{F}$ if $\mathbb{F}$ is a subfield of $\mathbb{F}'$, that is, $\mathbb{F}$ is a subset of $\mathbb{F}'$ and restricting the addition and multiplication laws of $\mathbb{F}'$ to the subset $\mathbb{F}$ recovers the appropriate laws of $\mathbb{F}$.

Now it can be shown that whenever $p \in \mathbb{P}$ is a prime and $m \in \mathbb{N}$ a natural number, then there is a field $\mathbb{F}_{p^m}$ with characteristic p and p^m elements such that $\mathbb{F}_{p^m}$ is an extension field of the prime field $\mathbb{F}_p$.

Similarly to the way prime fields $\mathbb{F}_p$ are generated by starting with the ring of integers and then dividing by a prime number p and keeping the remainder, prime field extensions $\mathbb{F}_{p^m}$ are generated by starting with the ring $\mathbb{F}_p[x]$ of polynomials and then dividing them by an irreducible polynomial of degree m and keeping the remainder.

To be more precise, let $P \in F_p[x]$ be an irreducible polynomial of degree m with coefficients from the given prime field $\mathbb{F}_p$. Then the underlying set $\mathbb{F}_{p^m}$ of the extension field is given by the set of all polynomials with a degree less then m:

$$\mathbb{F}_{p^m} := \{a_{m-1}x^{m-1} + a_{k-2}x^{k-2} + \ldots + a_1 x + a_0 \mid a_i \in \mathbb{F}_p\} \tag{4.29}$$

This can be shown to be the set of all remainders when dividing any polynomial $Q \in \mathbb{F}_p[x]$ by P, consequently, elements of the extension field are polynomials of degree less than m. This is analogous to how $\mathbb{F}_p$ is the set of all remainders when dividing integers by p.

Addition is inherited from $\mathbb{F}_p[x]$, which means that addition on $\mathbb{F}_{p^m}$ is defined like normal addition of polynomials:

$$+ : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \; , \; (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \sum_{j=0}^{m} (a_j + b_j) x^j \tag{4.30}$$

We can see that the neutral element is (the polynomial) 0, and that the additive inverse is given by the polynomial with all negative coefficients.

Multiplication is inherited from $\mathbb{F}_p[x]$, too, but we have to divide the result by our modulus polynomial P whenever the degree of the resulting polynomial is equal or greater to m:

$$\cdot \; \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \; , \; (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \left(\sum_{n=0}^{2m} \sum_{i=0}^{n} a_i b_{n-i} x^n \right) \bmod P \tag{4.31}$$

We can see that the neutral element is (the polynomial) 1. It is, however, not obvious from this definition how the multiplicative inverse looks.

We can easily see from the definition of $\mathbb{F}_{p^m}$ that the field is of characteristic p, since the multiplicative neutral element 1 is equivalent to the multiplicative element 1 from the underlying prime field, and hence $\sum_{j=0}^{p} 1 = 0$. Moreover, $\mathbb{F}_{p^m}$ is finite and contains p^m many elements, since elements are polynomials of degree $< m$, and every coefficient a_j can have a p number of different values. In addition, we see that the prime field $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p^m}$ that occurs when we restrict the elements of $\mathbb{F}_p$ to polynomials of degree zero.

One key point is that the construction of $\mathbb{F}_{p^m}$ depends on the choice of an irreducible polynomial, and, in fact, different choices will give different multiplication tables, since the remainders from dividing a product by P will be different.

It can, however, be shown that the fields for different choices of P are **isomorphic**, which means that there is a one-to-one correspondence between all of them. Consequently, from an abstract point of view, they are the same thing. From an implementations point of view, however, some choices are preferable to others because they allow for faster computations.

To summarize, we have seen that when a prime field $\mathbb{F}_p$ is given, any field $\mathbb{F}_{p^m}$ constructed in the above manner is a field extension of $\mathbb{F}_p$. To be more general, a field $\mathbb{F}_{p^{m_2}}$ is a field extension of a field $\mathbb{F}_{p^{m_1}}$, if and only if m_1 divides m_2. From this, we can deduce that, for any given fixed prime number, there are nested sequences of fields whenever the power m_j divides the power m_{j+1}, such that $\mathbb{F}_{p^{m_j}}$ is a subfield of $\mathbb{F}_{p^{m_{j+1}}}$:

$$\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}} \tag{4.32}$$

To get a more intuitive picture of this, we construct an extension field of the prime field $\mathbb{F}_3$ in the following example, and we can see how $\mathbb{F}_3$ sits inside that extension field.

Example 63 (The Extension field $\mathbb{F}_{3^2}$). In (XXX) we have constructed the prime field $\mathbb{F}_3$. In this example, we apply the definition of a field extension (page 64) to construct $\mathbb{F}_{3^2}$. We start by choosing an irreducible polynomial of degree 2 with coefficients in $\mathbb{F}_3$. We try $P(t) = t^2 + 1$. Possibly the fastest way to show that P is indeed irreducible is to just insert all elements from

add reference

check reference

$\mathbb{F}_3$ to see if the result is ever zero. We compute as follows:

$$
\begin{aligned}
P(0) &= 0^2 + 1 = 1 \\
P(1) &= 1^2 + 1 = 2 \\
P(2) &= 2^2 + 1 = 1 + 1 = 2
\end{aligned}
$$

This implies that P is irreducible. The set $\mathbb{F}_{3^2}$ contains all polynomials of degrees lower than two, with coefficients in $\mathbb{F}_3$, which are precisely as listed below:

$$\mathbb{F}_{3^2} = \{0, 1, 2, t, t+1, t+2, 2t, 2t+1, 2t+2\}$$

As expected, our extension field contains 9 elements. Addition is defined as addition of polynomials; for example $(t+2) + (2t+2) = (1+2)t + (2+2) = 1$. Doing this computation for all elements gives the following addition table

+	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
1	1	2	0	t+1	t+2	t	2t+1	2t+2	2t
2	2	0	1	r+2	t	t+1	2t+2	2t	2t+1
t	t	t+1	t+2	2t	2t+1	2t+2	0	1	2
t+1	t+1	t+2	t	2t+1	2t+2	2t	1	2	0
t+2	t+2	t	t+1	2t+2	2t	2t+1	2	0	1
2t	2t	2t+1	2t+2	0	1	2	t	t+1	t+2
2t+1	2t+1	2t+2	2t	1	2	0	t+1	t+2	t
2t+2	2t+2	2t	2t+1	2	0	1	t+2	t	t+1

As we can see, the group $(\mathbb{F}_3, +)$ is a subgroup of the group $(\mathbb{F}_{3^2}, +)$, obtained by only considering the first three rows and columns of this table.

As it was the case in previous examples, we can use the table to deduce the negative of any element from $\mathbb{F}_{3^2}$. For example, in $\mathbb{F}_{3^2}$ we have $-(2t+1) = t+2$, since $(2t+1) + (t+2) = 0$

Multiplication needs a bit more computation, as we first have to multiply the polynomials, and whenever the result has a degree ≥ 2, we have to divide it by P and keep the remainder. To see how this works, let us compute the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$:

$$
\begin{aligned}
(t+2)\cdot(2t+2) &= (2t^2 + 2t + t + 1) \bmod (t^2+1) \\
&= (2t^2 + 1) \bmod (t^2+1) && \#\, 2t^2+1 : t^2+1 = 2 + \frac{2}{t^2+1} \\
&= 2
\end{aligned}
$$

This means that the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$ is 2. Performing this computation for all elements gives the following multiplication table:

·	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
0	0	0	0	0	0	0	0	0	0
1	0	1	2	t	t+1	t+2	2t	2t+1	2t+2
2	0	2	1	2t	2t+2	2t+1	t	t+2	t+1
t	0	t	2t	2	t+2	2t+2	1	t+1	2t+1
t+1	0	t+1	2t+2	t+2	2t	1	2t+1	2	t
t+2	0	t+2	2t+1	2t+2	1	t	t+1	2t	2
2t	0	2t	t	1	2t+1	t+1	2	2t+2	t+2
2t+1	0	2t+1	t+2	t+1	2	2t	2t+2	t	1
2t+2	0	2t+2	t+1	2t+1	t	2	t+2	1	2t

As it was the case in previous examples, we can use the table to deduce the multiplicative inverse of any non-zero element from $\mathbb{F}_{3^2}$. For example, in $\mathbb{F}_{3^2}$ we have $(2t+1)^{-1} = 2t+2$, since $(2t+1)\cdot(2t+2) = 1$.

From the multiplication table, we can also see that the only quadratic residues in $\mathbb{F}_{3^2}$ are from the set $\{0,1,2,t,2t\}$, with $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,2\}$, $\sqrt{2} = \{t,2t\}$, $\sqrt{t} = \{t+2,2t+1\}$ and $\sqrt{2t} = \{t+1,2t+2\}$.

Since $\mathbb{F}_{3^2}$ is a field, we can solve equations as we would for other fields, (such as rational numbers). To see that, let us find all $x \in \mathbb{F}_{3^2}$ that solve the quadratic equation $(t+1)(x^2+(2t+2)) = 2$. We compute as follows:

$$\begin{aligned}
(t+1)(x^2+(2t+2)) &= 2 && \text{\# 2 distributive law}\\
(t+1)x^2+(t+1)(2t+2) &= 2\\
(t+1)x^2+(t) &= 2 && \text{\# 2 add the additive inverse of } t\\
(t+1)x^2+(t)+(2t) &= (2)+(2t)\\
(t+1)x^2 &= 2t+2 && \text{\# multiply with the multiplicative invers of } t+1\\
(t+2)(t+1)x^2 &= (t+2)(2t+2) && \text{\# multiply with the multiplicative invers of } t+1\\
x^2 &= 2 && \text{\# 2 is quadratic residue. Take the roots.}\\
x &\in \{t,2t\}
\end{aligned}$$

Computations in extension fields are arguably on the edge of what can reasonably be done with pen and paper. Fortunately, Sage provides us with a simple way to do the computations.

```
sage: Z3 = GF(3) # prime field
sage: Z3t.<t> = Z3[] # polynomials over Z3
sage: P = Z3t(t^2+1)
sage: P.is_irreducible()
True
sage: F3_2.<t> = GF(3^2, name='t', modulus=P)
sage: F3_2
Finite Field in t of size 3^2
sage: F3_2(t+2)*F3_2(2*t+2) == F3_2(2)
True
sage: F3_2(2*t+2)^(-1) # multiplicative inverse
2*t + 1
sage: # verify our solution to (t+1)(x^2 + (2t+2)) = 2
sage: F3_2(t+1)*(F3_2(t)**2 + F3_2(2*t+2)) == F3_2(2)
True
sage: F3_2(t+1)*(F3_2(2*t)**2 + F3_2(2*t+2)) == F3_2(2)
True
```

Exercise 36. Consider the extension field $\mathbb{F}_{3^2}$ from the previous example and find all pairs of elements $(x,y) \in \mathbb{F}_{3^2}$, for which the following equation holds:

$$y^2 = x^3+4 \tag{4.33}$$

Exercise 37. Show that the polynomial $P = x^3+x+1$ from $\mathbb{F}_5[x]$ is irreducible. Then consider the extension field $\mathbb{F}_{5^3}$ defined relative to P. Compute the multiplicative inverse of $(2t^2+4) \in$

$\mathbb{F}_{5^3}$ using the extended Euclidean algorithm. Then find all $x \in \mathbb{F}_{5^3}$ that solve the following equation:

$$(2t^2+4)(x-(t^2+4t+2)) = (2t+3) \tag{4.34}$$

Hashing into extension fields On page 64, we have seen how to hash into prime fields. As elements of extension fields can be seen as polynomials over prime fields, hashing into extension fields is therefore possible if every coefficient of the polynomial is hashed independently. check reference

4.4 Projective Planes

Projective planes are particular geometric constructs defined over a given field. In a sense, projective planes extend the concept of the ordinary Euclidean plane by including "points at infinity."

Such an inclusion of infinity points makes projective planes particularly useful in the description of elliptic curves, as the description of such a curve in an ordinary plane needs an additional symbol "the point at infinity" to give the set of points on the curve the structure of a group. Translating the curve into projective geometry includes this "point at infinity" more naturally into the set of all points on a projective plane.

To understand the idea of constructing of projective planes, note that in an ordinary Euclidean plane, two lines either intersect in a single point or are parallel. In the latter case, both lines are either the same, that is, they intersect in all points, or do not intersect at all. A projective plane can then be thought of as an ordinary plane, but equipped with additional "point at infinity" such that two different lines always intersect in a single point. Parallel lines intersect "at infinity".

To be more precise, let $\mathbb{F}$ be a field, $\mathbb{F}^3 := \mathbb{F}\times\mathbb{F}\times F$ the set of all three tuples over $\mathbb{F}$ and $x \in \mathbb{F}^3$ with $x = (X,Y,Z)$. Then there is exactly one *line* in $\mathbb{F}^3$ that intersects both $(0,0,0)$ and x. This line is given as follows:

$$[X:Y:Z] := \{(k\cdot X, k\cdot Y, k\cdot Z) \mid k \in \mathbb{F}\} \tag{4.35}$$

A **point** in the **projective plane** over $\mathbb{F}$ is defined as such a **line**, and the projective plane is the set of all such points:

$$\mathbb{FP}^2 := \{[X:Y:Z] \mid (X,Y,Z) \in \mathbb{F}^3 \text{ with } (X,Y,Z) \neq (0,0,0)\} \tag{4.36}$$

It can be shown that a projective plane over a finite field $\mathbb{F}_{p^m}$ contains $p^{2m}+p^m+1$ number of elements.

To understand why $[X:Y:Z]$ is called a line, consider the situation where the underlying field $\mathbb{F}$ is the set of real numbers $\mathbb{R}$. In this case, $\mathbb{R}^3$ can be seen as the three-dimensional space, and $[X:Y:Z]$ is an ordinary line in this 3-dimensional space that intersects zero and the point with coordinates X, Y and Z.

The key observation here is that points in the projective plane are lines in the 3-dimensional space $\mathbb{F}^3$,]. Additionally, for finite fields, the terms **space** and **line** share very little visual similarity with their counterparts over the set of real numbers.

It follows from this that points $[X:Y:Z] \in \mathbb{FP}^2$ are not simply described by fixed coordinates (X,Y,Z), but by **sets of coordinates**, where two different coordinates (X_1,Y_1,Z_1) and (X_2,Y_2,Z_2) describe the same point if and only if there is some field element k such that $(X_1,Y_1,Z_1) = (k\cdot X_2, k\cdot Y_2, k\cdot Z_2)$. Points $[X:Y:Z]$ are called **projective coordinates**.

Notation and Symbols 6 (Projective coordinates). Projective coordinates of the form $[X : Y : 1]$ are descriptions of so-called **affine points**. Projective coordinates of the form $[X : Y : 0]$ are descriptions of so-called **points at infinity**. In particular, the projective coordinate $[1 : 0 : 0]$ describes the so-called **line at infinity**.

Example 64. Consider the field $\mathbb{F}_3$ from example XXX. As this field only contains three elements, it does not take too much effort to construct its associated projective plane $\mathbb{F}_3\mathbb{P}^2$, as we know that it only contains 13 elements.

add reference

To find $\mathbb{F}_3\mathbb{P}^2$, we have to compute the set of all lines in $\mathbb{F}_3 \times \mathbb{F}_3 \times \mathbb{F}_3$ that intersect $(0,0,0)$. Since those lines are parameterized by tuples (x_1,x_2,x_3), we compute as follows:

$$
\begin{aligned}
{[0:0:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,0,1),(0,0,2)\}\\
{[0:0:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,0,2),(0,0,1)\} = [0:0:1]\\
{[0:1:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,1,0),(0,2,0)\}\\
{[0:1:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,1,1),(0,2,2)\}\\
{[0:1:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,1,2),(0,2,1)\}\\
{[0:2:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,2,0),(0,1,0)\} = [0:1:0]\\
{[0:2:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,2,1),(0,1,2)\} = [0:1:2]\\
{[0:2:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(0,2,2),(0,1,1)\} = [0:1:1]\\
{[1:0:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,0,0),(2,0,0)\}\\
{[1:0:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,0,1),(2,0,2)\}\\
{[1:0:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,0,2),(2,0,1)\}\\
{[1:1:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,1,0),(2,2,0)\}\\
{[1:1:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,1,1),(2,2,2)\}\\
{[1:1:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,1,2),(2,2,1)\}\\
{[1:2:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,2,0),(2,1,0)\}\\
{[1:2:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,2,1),(2,1,2)\}\\
{[1:2:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(1,2,2),(2,1,1)\}\\
{[2:0:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,0,0),(1,0,0)\} = [1:0:0]\\
{[2:0:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,0,1),(1,0,2)\} = [1:0:2]\\
{[2:0:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,0,2),(1,0,1)\} = [1:0:1]\\
{[2:1:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,1,0),(1,2,0)\} = [1:2:0]\\
{[2:1:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,1,1),(1,2,2)\} = [1:2:2]\\
{[2:1:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,1,2),(1,2,1)\} = [1:2:1]\\
{[2:2:0]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,2,0),(1,1,0)\} = [1:1:0]\\
{[2:2:1]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,2,1),(1,1,2)\} = [1:1:2]\\
{[2:2:2]} &= \{(k\cdot x_1, k\cdot x_2, k\cdot x_3) \mid k \in \mathbb{F}_3\} = \{(2,2,2),(1,1,1)\} = [1:1:1]
\end{aligned}
$$

These lines define the 13 points in the projective plane $\mathbb{F}_3\mathbb{P}$:

$$
\begin{aligned}
\mathbb{F}_3\mathbb{P} = \{&[0:0:1],[0:1:0],[0:1:1],[0:1:2],[1:0:0],[1:0:1],\\
&[1:0:2],[1:1:0],[1:1:1],[1:1:2],[1:2:0],[1:2:1],[1:2:2]\}
\end{aligned}
$$

This projective plane contains 9 affine points, three points at infinity and one line at infinity.

To understand the ambiguity in projective coordinates a bit better, let us consider the point $[1:2:2]$. As this point in the projective plane is a line in $\mathbb{F}_3^3$, it has the projective coordinates $(1,2,2)$ as well as $(2,1,1)$, since the former coordinate gives the latter when multiplied in $\mathbb{F}_3$ by the factor 2. In addition, note that, for the same reasons, the points $[1:2:2]$ and $[2:1:1]$ are the same, since their underlying sets are equal.

Exercise 38. Construct the so-called **Fano plane**, that is, the projective plane over the finite field $\mathbb{F}_2$.

Chapter 5

Elliptic Curves

TODO: Elliptic Curve asymmetric cryptography examples. Private key, generator, public key.

Generally speaking, elliptic curves are "curves" defined in geometric planes like the Euclidean or the projective plane over some given field. One of the key features of elliptic curves over finite fields from the point of view of cryptography is that their set of points has a group law such that the resulting group is finite and cyclic, and it is believed that the discrete logarithm problem on these groups is hard.

A special class of elliptic curves are so-called **pairing-friendly curves**, which have a notation of a group pairing as defined in XXX. This pairing has cryptographically advantageous properties. Those curve are useful in the development of SNARKs, since they allow to compute so-called R1CS-satisfiability "in the exponent" MIRCO: (THIS HAS TO BE REWRITTEN WITH WAY MORE DETAIL)

add reference

In this chapter, we introduce epileptic curves as they are used in pairing-based approaches to the construction of SNARKs. The elliptic curves we consider are all defined over prime fields or prime field extensions and the reader should be familiar with the contend of the previous section on those fields.

In its most generality elliptic curves are defined as a smooth projective curve of genus 1 defined over some field $\mathbb{F}$ with a distinguished $\mathbb{F}$-rational point, but this definition is not very useful for the introductory character of this book. We will therefore look at 3 more practical definitions in the following sections, by introducing Weierstraß, Montgomery and Edwards curves. All of them are widely used in cryptography, and understanding them is crucial to being able to follow the rest of this book.

maybe remove this sentence?

5.1 Elliptic Curve Arithmetics

5.1.1 Short Weierstraß Curves

In this section, we introduce **short Weierstraß** curves, which are the most general types of curves over finite fields of characteristic greater than 3.

We start with their representation in affine space. This representation has the advantage that affine points correspond to pairs of numbers, which makes it more accessible for beginners. However, it has the disadvantage that a special "point at infinity", that is not a point on the curve, is necessary to describe the group structure. We introduce the elliptic curve group law and describe elliptic curve scalar multiplication, which is an instantiation of the exponential map from general cyclic groups.

affine space

Then we look at the projective representation of short Weierstraß curves. This has the advantage that no special symbol is necessary to represent the point at infinity but comes with the drawback that projective points are classes of numbers, which might be a bit unusual for a beginner.

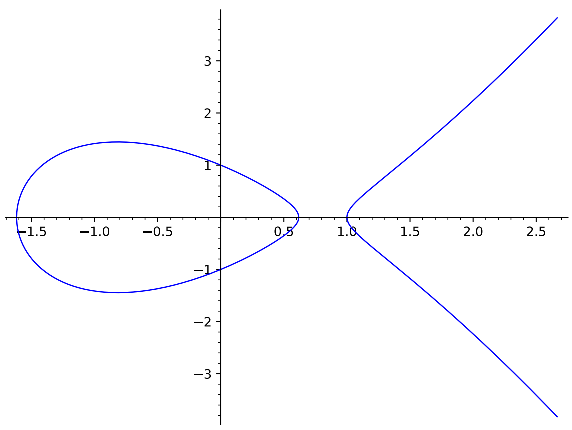
We finish this section with an explicit equivalence that transforms affine representations into projective ones and vice versa.

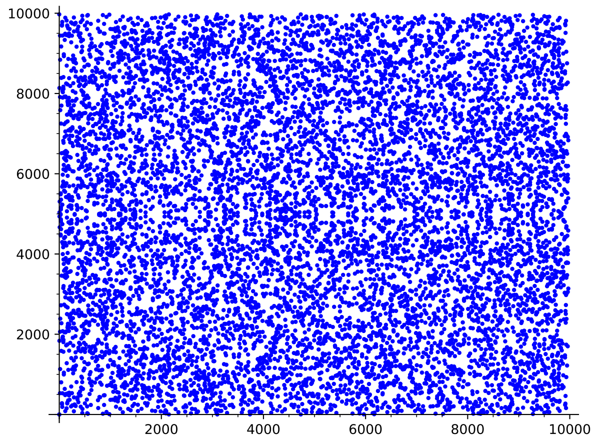
Affine short Weierstraß form Probably the least abstract and most straight-forward way to introduce elliptic curves for non-mathematicians and beginners is the so-called affine representation of a short Weierstraß curve. To see what this is, let $\mathbb{F}$ be a finite field of order q and $a, b \in \mathbb{F}$ two field elements such that $4a^3 + 27b^2 \bmod q \neq 0$. Then a **short Weierstraß elliptic curve** $E(\mathbb{F})$ over $\mathbb{F}$ in its affine representation is the set of all pairs of field elements $(x, y) \in \mathbb{F} \times \mathbb{F}$ that satisfy the short Weierstraß cubic equation $y^2 = x^3 + a \cdot x + b$, together with a distinguished symbol $\mathcal{O}$, called the **point at infinity**:

$$E(\mathbb{F}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + a \cdot x + b\} \bigcup \{\mathcal{O}\} \tag{5.1}$$

Notation and Symbols 7. In the literature, the set $E(\mathbb{F})$, which includes the symbol $\mathcal{O}$, is often called the set of **rational points** of the elliptic curve, in which case the curve itself is usually written as $E/\mathbb{F}$. However, in what follows, we will frequently identify an elliptic curve with its set of rational points and therefore use the notation $E(\mathbb{F})$ instead. This is possible in our case, since we only the group structure of the curve in consideration is relevant for us.

The term "curve" is used here because, in the ordinary 2 dimensional plane $\mathbb{R}^2$, the set of all points (x, y) that satisfy $y^2 = x^3 + a \cdot x + b$ looks like a curve. We should note however that visualizing elliptic curves over finite fields as "curves" has its limitations, and we will therefore not stress the geometric picture too much, but focus on the computational properties instead. To understand the visual difference, consider the following two elliptic curves:





Both elliptic curves are defined by the same short Weierstraß equation $y^2 = x^3 - 2x + 1$, but the first curve is defined in the real affine plane $\mathbb{R}^2$, that is, the pair (x, y) contains real numbers, while the second one is defined in the affine plane $\mathbb{F}_{9973}^2$, which means that both x and y are from the prime field $\mathbb{F}_{9973}$. Every blue dot represents a pair (x, y), that is a solution to $y^2 = x^3 - 2x + 1$. As we can see, the second curve hardly looks like a geometric structure one would naturally call a curve. This shows that our geometric intuitions from $\mathbb{R}^2$ are obfuscated in curves over finite fields.

The identity $6 \cdot (4a^3 + 27b^2) \bmod q \neq 0$ ensures that the curve is non-singular, which basically means that the curve has no cusps or self-intersections.

cusps

self-intersections

Throughout this book, the reader is advised to do as many computations in a pen-and-paper fashion as possible, as this is helps getting a deeper understanding of the details. However, when dealing with elliptic curves, computations can quickly become cumbersome and tedious, and one might get lost in the details. Fortunately, Sage is very helpful in dealing with elliptic curves. This book to introduces the reader to the great elliptic curve capabilities of Sage. The following snippet shows a way to define elliptic curves and work with them in Sage:

```
sage: F5 = GF(5) # define the base field
sage: a = F5(2) # parameter a
sage: b = F5(4) # parameter b
sage: # check non-sigularity
sage: F5(6)*(F5(4)*a^3+F5(27)*b^2) != F5(0)
True
sage: # short Weierstrass curve
sage: E = EllipticCurve(F5,[a,b]) # y^2 == x^3 + ax +b
sage: P = E(0,2) # 2^2 == 0^3 + 2*0 + 4
sage: P.xy() # affine coordinates
(0, 2)
sage: INF = E(0) # point at infinity
sage: try:  # point at infinity has no affine coordinates
....:     INF.xy()
....: except ZeroDivisionError:
....:     pass
sage: P = E.plot() # create a plotted version
```

The following three examples give a more practical understanding of what an elliptic curve is and how we can compute it. The reader is advised to read them carefully, and ideally, to also carry out the computation themselves. We will repeatedly build on these examples in this chapter, and use the second example throughout this book.

Example 65. To provide the reader with an example of a small elliptic curve where all computation can be done with pen and paper, consider the prime field $\mathbb{F}_5$ from example 59 (page 62), which should be quite familiar to readers who had worked through the examples and exercises in the previous chapter.

check reference

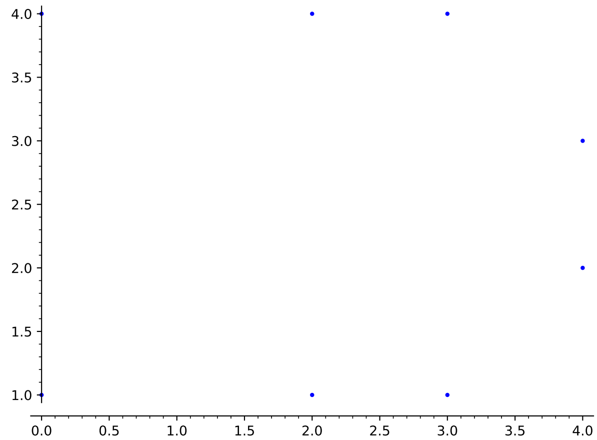
To define an elliptic curve over $\mathbb{F}_5$, we have to choose to numbers a and b from that field. Assuming we choose $a = 1$ and $b = 1$ then $4a^3 + 27b^2 \equiv 1 \pmod{5}$ from which follows that the corresponding elliptic curve $E_1(\mathbb{F}_5)$ is given by the set of all pairs (x,y) from $\mathbb{F}_5$ that satisfy the equation $y^2 = x^3 + x + 1$, together with the special symbol $\mathcal{O}$, which represents the "point at infinity".

To get a better understand of that curve, observer that if we choose arbitrarily the pair $(x,y) = (1,1)$, we see that $1^2 \neq 1^3 + 1 + 1$ and hence $(1,1)$ is not an element of the curve $E_1(\mathbb{F}_5)$. On the other hand choosing for example $(x,y) = (2,1)$ gives $1^2 = 2^3 + 2 + 1$ and hence the pair $(2,1)$ is an element of $E_1(\mathbb{F}_5)$ (Remember that all computations are done in modulo 5 arithmetics).

Now since the set $\mathbb{F}_5 \times \mathbb{F}_5$ of all pairs (x,y) from $\mathbb{F}_5$ contains only $5 \cdot 5 = 25$ pairs, we can compute the curve, by just inserting every possible pair (x,y) into the short Weierstraß equation $y^2 = x^3 + x + 1$. If the equation holds, the pair is a curve point, if not that means that the point is not on the curve. Combining the result of this computation with the point at infinity gives the curve as follows:

$$E_1(\mathbb{F}_5) = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\}$$

This means that our elliptic curve is a set of 9 elements, 8 of which are pairs of numbers and one special symbol $\mathcal{O}$. Visualizing $E1$ gives the following plot:



In the development of SNARKs, it is sometimes necessary to do elliptic curve cryptography “in a circuit", which basically means that the elliptic curves need to be implemented in a certain SNARK-friendly way. We will look at what this means in chapter 7. To be able to do this efficiently, it is desirable to have curves with special properties. The following example is a pen-and-paper version of such a curve, called **Baby-jubjub**, which resembles cryptographically secure curves extensively used in real-world SNARKs. The interested reader is advised to study this example carefully, as we will use it and build on it in various places throughout the book. S: I feel like a lot of people won’t get the Lewis Carroll reference unless we make it more explicit. M: The term Baby-JubJub is actually the name of a curve used in zCash and Ethereum a lot. IDK why they choosed that name.

check reference

jubjub

Example 66 (Tiny-Jubjub). Consider the prime field $\mathbb{F}_{13}$ from exercise 4.3 (page 64. If we choose $a = 8$ and $b = 8$, then $4a^3 + 27b^2 \equiv 6 \pmod{13}$ and the corresponding elliptic curve is given by all pairs (x, y) from $\mathbb{F}_{13}$ such that $y^2 = x^3 + 8x + 8$ holds. We call this curve the **Tiny-jubjub** curve, or *TJJ_13* for short.

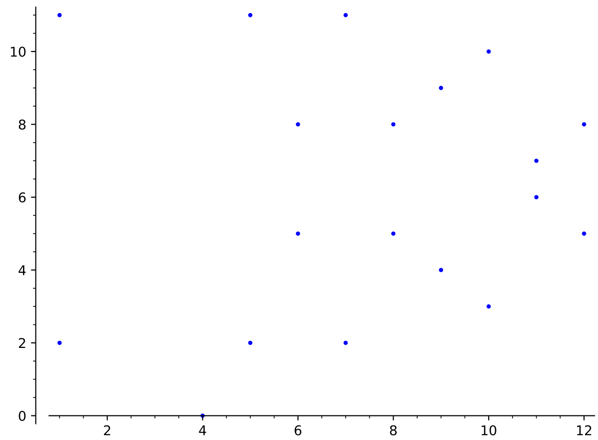
check reference

Now, since the set $\mathbb{F}_{13} \times \mathbb{F}_{13}$ of all pairs (x, y) from $\mathbb{F}_{13}$ contains only $13 \cdot 13 = 169$ pairs, we can compute the curve by just inserting every possible pair (x, y) into the short Weierstraß equation $y^2 = x^3 + 8x + 8$. We get the following result:

$$
\begin{aligned}
TJJ\_13 = \{&\mathcal{O}, (1,2), (1,11), (4,0), (5,2), (5,11), (6,5), (6,8), (7,2), (7,11), (8,5), (8,8), \\
&(9,4), (9,9), (10,3), (10,10), (11,6), (11,7), (12,5), (12,8)\}
\end{aligned}
\tag{5.2}
$$

As we can see, the curve consists of 20 points; 19 points from the affine plane and the point at infinity. To get a visual impression of the *TJJ_13* curve, we might plot all of its points (except the point at infinity) in the $\mathbb{F}_{13} \times \mathbb{F}_{13}$ affine plane. We get the following plot:

affine plane



As we will see in what follows, this curve is rather special, as it is possible to represent it in two alternative forms called the **Montgomery** and the **twisted Edwards form** (See sections 5.1.2 and 5.1.3, respectively).

check reference

check reference

Now that we have seen two pen-and-paper friendly elliptic curves, let us look at a curve, that is used in actual cryptography. Cryptographically secure elliptic curves are not **qualitatively** different from the curves we looked at so far, but the prime number modulus of their prime field is much larger. Typical examples use prime numbers that have binary representations in the magnitude of more than double the size of the desired security level. If, for example, a security of 128 bits is desired, a prime modulus of binary size ≥ 256 is chosen. The following example provides such a curve.

Example 67 (Bitcoin's Secp256k1 curve). To give an example of a real-world, cryptographically secure curve, let us look at curve Secp256k1, which is famous for being used in the public key cryptography of Bitcoin. The prime field $\mathbb{F}_p$ of Secp256k1 is defined by the following prime number:

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

The binary representation of this number needs 256 bits, which implies that the prime field $\mathbb{F}_p$ contains approximately 2^{256} many elements, which is considered quite large. To get a better impression of how large the base field is, consider that the number 2^{256} is approximately in the same order of magnitude as the estimated number of atoms in the observable universe.

The curve Secp256k1 is defined by the parameters $a, b \in \mathbb{F}_p$ with $a = 0$ and $b = 7$. Since $4 \cdot 0^3 + 27 \cdot 7^2 \bmod p = 1323$, those parameters indeed define an elliptic curve given as follows:

$$Secp256k1 = \{(x,y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + 7\}$$

Clearly, the Secp256k1 curve is too large to do computations by hand, since it can be shown that the number of its elements is a prime number r that also has a binary representation of 256 bits:

$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

Cryptographically secure elliptic curves are therefore not useful in pen-and-paper computations. Fortunately, Sage handles large curves efficiently:

```
sage: p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
```

```
sage: # Hexadecimal representation
sage: p.str(16)
fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc
  2f
sage: p.is_prime()
True
sage: p.nbits()
256
sage: Fp = GF(p)
sage: Secp256k1 = EllipticCurve(Fp,[0,7])
sage: r = Secp256k1.order() # number of elements
sage: r.str(16)
fffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd03641
  41
sage: r.is_prime()
True
sage: r.nbits()
256
```

Exercise 39. Look up the definition of curve BLS12-381, implement it in Sage and compute its order.

Affine compressed representation As we have seen in example 67, cryptographically secure elliptic curves are defined over large prime fields, where elements of those fields typically need more than 255 bits of storage on a computer. Since elliptic curve points consist of pairs of those field elements, they need double that amount of storage.

check reference

However, we can reduce the amount of space needed to represent a curve point by using a technique called **point compression**. Note that, up to a sign, the y coordinate of a curve point can be computed from the x coordinate by simply inserting x into the Weierstraß equation and then computing the roots of the result. This gives two results, and it means that we can represent a curve point in **compressed form** by simply storing the x coordinate together with a single sign bit only, the latter of which deterministically decides which of the two roots to choose. One convention could be to always choose the root closer to 0 when the sign bit is 0, and the root closer to the order of $\mathbb{F}$ when the sign bit is 1. In case the y coordinate is zero, both sign bits give the same result.

sign

more explanation of what the sign is

Example 68 (Tiny-jubjub). To understand the concept of compressed curve points a bit better, consider the *TJJ_13* curve from example 66 again. Since this curve is defined over the prime field $\mathbb{F}_{13}$, and numbers between 0 and 13 need approximately 4 bits to be represented, each *TJJ_13* point on this curve needs 8 bits of storage in uncompressed form. The following set represents the uncompressed form of the points on this curve:

check reference

$$
\begin{aligned}
TJJ\_13 = \{&\mathcal{O},(1,2),(1,11),(4,0),(5,2),(5,11),(6,5),(6,8),(7,2),(7,11),\\
&(8,5),(8,8),(9,4),(9,9),(10,3),(10,10),(11,6),(11,7),(12,5),(12,8)\}
\end{aligned}
$$

Using the technique of point compression, we can reduce the bits needed to represent the points on this curve to 5 per point. To achieve this, we can replace the y coordinate in each (x,y) pair by a sign bit indicating whether or not y is closer to 0 or to 13. As a result y values in the range $[0,\ldots,6]$ will have the sign bit 0, while y-values in the range $[7,\ldots,12]$ will have the sign bit 1.

Applying this to the points in *TJJ_13* gives the compressed representation as follows:

$$TJJ\_13 = \{\mathcal{O}, (1,0), (1,1), (4,0), (5,0), (5,1), (6,0), (6,1), (7,0), (7,1), \\ (8,0), (8,1), (9,0), (9,1), (10,0), (10,1), (11,0), (11,1), (12,0), (12,1)\}$$

Note that the numbers $7, \ldots, 12$ are the negatives (additive inverses) of the numbers $1, \ldots, 6$ in modular 13 arithmetics and that $-0 = 0$. Calling the compression bit a "sign bit" therefore makes sense.

S: I don't follow this at all

To recover the uncompressed counterpart of, say, the compressed point $(5,1)$, we insert the x coordinate 5 into the Weierstraß equation and get $y^2 = 5^3 + 8 \cdot 5 + 8 = 4$. As expected, 4 is a quadratic residue in $\mathbb{F}_{13}$ with roots $\sqrt{4} = \{2, 11\}$. Since the sign bit of the point is 1, we have to choose the root closer to the modulus 13, which is 11. The uncompressed point is therefore $(5, 11)$.

Looking at the previous examples, the compression rate does not look very impressive. However, looking at the real-life example of the Secp256k1 curve shows that compression is has significant practical advantages.

Example 69. Consider the Secp256k1 curve from example 67 again. The following code invokes Sage to generate a random affine curve point, then applies our compression method to it:

check reference

```
sage: P = Secp256k1.random_point().xy()
sage: P
(25466686146381466796099675229731837307656890594507867081905973445487613678674, 53433031093696009811691078630770548173153006157646869877613522266300751231219)
sage: # uncompressed affine point size
sage: ZZ(P[0]).nbits()+ZZ(P[1]).nbits()
509
sage: # compute the compression
sage: if P[1] > Fp(-1)/Fp(2):
....:     PARITY = 1
....: else:
....:     PARITY = 0
sage: PCOMPRESSED = [P[0],PARITY]
sage: PCOMPRESSED
[25466686146381466796099675229731837307656890594507867081905973445487613678674, 0]
sage: # compressed affine point size
sage: ZZ(PCOMPRESSED[0]).nbits()+ZZ(PCOMPRESSED[1]).nbits()
254
```

add explanation of how this shows what we claim

Affine group law One of the key properties of an elliptic curve is that it is possible to define a group law on the set of its rational points such that the point at infinity serves as the neutral element and inverses are reflections on the x-axis.

The origin of this law can be understood in a geometric picture and is known as the **chord-and-tangent rule**. In the affine representation of a short Weierstraß curve, the rule can be described in the following way:

Definition 5.1.1.1. **Chord-and-tangent rule**

- (Point at infinity) We define the point at infinity $\mathcal{O}$ as the neutral element of addition, that is, we define $P + \mathcal{O} = P$ for all points $P \in E(\mathbb{F})$. should this def. be moved even earlier?
- (Point addition) Let $P,Q \in E(\mathbb{F})\backslash\{\mathcal{O}\}$ with $P \neq Q$ be two distinct points on an elliptic curve, neither of them the point at infinity. The sum of P and Q is defined as follows: Consider the line l which intersects the curve in P and Q. If l intersects the elliptic curve at a third point R', define the sum $R = P \oplus Q$ of P and Q as the reflection of R' at the x-axis. If the line l does not intersect the curve at a third point, define the sum to be the point at infinity $\mathcal{O}$. It can be shown that no such chord line will intersect the curve in more than three points, so addition is not ambiguous. chord line
- (Point doubling) Let $P \in E(\mathbb{F})\backslash\{\mathcal{O}\}$ be a point on an elliptic curve, that is not the point at infinity. The sum of P with itself (the doubling of P) is defined as follows: Consider the line which is tangential to the elliptic curve at P. If this line intersects the elliptic curve at a second point R', the sum $2P = P + P$ is the reflection of R' at the x-axis. If it does not intersect the curve at a third, point define the sum to be the point at infinity $\mathcal{O}$. It can be shown that no such tangent line will intersect the curve in more than two points, so addition is not ambiguous. tangential tangent line

It can be shown that the points of an elliptic curve form a commutative group with respect to the tangent-and-chord rule such that $\mathcal{O}$ acts the neutral element, and the inverse of any element $P \in E(\mathbb{F})$ is the reflection of P on the x-axis.

To translate the geometric description into algebraic equations, first observe that, for any two given curve points $(x_1, y_1), (x_2, y_2) \in E(\mathbb{F})$, it can be shown that the identity $x_1 = x_2$ implies $y_2 = \pm y_1$, which shows that the following rules are a complete description of the affine addition law.

Definition 5.1.1.2. **Chord-and-tangent rule: algebraic equations**

- (Neutral element) The point at infinity $\mathcal{O}$ is the neutral element.
- (Additive inverse) The additive inverse of $\mathcal{O}$ is $\mathcal{O}$. For any other curve point $(x,y) \in E(\mathbb{F})\backslash\{\mathcal{O}\}$, the additive inverse is given by $(x, -y)$.
- (Addition rule) For any two curve points $P,Q \in E(\mathbb{F})$, addition is defined by one of the following three cases:

  1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P \oplus Q = P$.
  2. (Adding inverse elements) If $P = (x,y)$ and $Q = (x,-y)$, the sum is defined as $P \oplus Q = \mathcal{O}$.
  3. (Adding non-self-inverse equal points) If $P = (x,y)$ and $Q = (x,y)$ with $y \neq 0$, the sum $2P = (x', y')$ is defined as follows: We only referred to P in the definition of point doubling above so Q seems a bit confusing here even though it's defined as equal to P remove Q?
  $$x' = \left(\frac{3x^2+a}{2y}\right)^2 - 2x \quad , \quad y' = \left(\frac{3x^2+a}{2y}\right)^2 (x - x') - y$$
  4. (Adding non-inverse different points) If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $x_1 \neq x_2$, the sum $R = P + Q$ with $R = (x_3, y_3)$ is defined as follows:
  $$x_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)^2 - x_1 - x_2 \quad , \quad y_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1 - x_3) - y_1$$

Note that short Weierstraß curve points P with $P = (x, 0)$ are inverses of themselves, which implies $2P = \mathcal{O}$ in this case.

Notation and Symbols 8. Let $\mathbb{F}$ be a field and $E(\mathbb{F})$ be an elliptic curve over $\mathbb{F}$. We write $\oplus$ for the group law on $E(\mathbb{F})$ and $(E(\mathbb{F}), \oplus)$ for the group of rational points.

As we can see, it is very efficient to compute inverses on elliptic curves. However, computing the addition of elliptic curve points in the affine representation needs to consider many cases and involves extensive finite field divisions. As we will see in the next paragraph, this can be simplified in projective coordinates. where?

To get some practical impression of how the group law on an elliptic curve is computed, let's look at some actual cases:

Example 70. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example 65 again. As we have seen, the set of rational points contains 9 elements: check reference

$$E_1(\mathbb{F}_5) = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\}$$

We know that this set defines a group, so we can add any two elements from $E_1(\mathbb{F}_5)$ to get a third element.

To give an example, consider the elements $(0,1)$ and $(4,2)$. Neither of these elements is the neutral element $\mathcal{O}$, and since, the x coordinate of $(0,1)$ is different from the x coordinate of $(4,2)$, we know that we have to use the chord rule, that is, rule number 4 from definition 5.1.1.2 to compute the sum $(0,1) \oplus (4,2)$: check reference

$$\begin{aligned}
x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 && \text{\# insert points} \\
&= \left(\frac{2-1}{4-0}\right)^2 - 0 - 4 && \text{\# simplify in } \mathbb{F}_5 \\
&= \left(\frac{1}{4}\right)^2 + 1 = 4^2 + 1 = 1 + 1 = 2
\end{aligned}$$

$$\begin{aligned}
y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 && \text{\# insert points} \\
&= \left(\frac{2-1}{4-1}\right)(0-2) - 1 && \text{\# simplify in } \mathbb{F}_5 \\
&= \left(\frac{1}{4}\right) \cdot 3 + 4 = 4 \cdot 3 + 4 = 2 + 4 = 1
\end{aligned}$$

So, in our elliptic curve $E_1(\mathbb{F}_5)$ we get $(0,1) \oplus (4,2) = (2,1)$, and, indeed, the pair $(2,1)$ is an element of $E_1(\mathbb{F}_5)$ as expected. On the other hand, $(0,1) \oplus (0,4) = \mathcal{O}$, since both points have equal x coordinates and inverse y coordinates, rendering them inverses of each other. Adding the point $(4,2)$ to itself, we have to use the tangent rule, that is, rule 3 from definition 5.1.1.2: check reference

$$
\begin{aligned}
x' &= \left(\frac{3x^2+a}{2y}\right)^2 - 2x && \text{\# insert points}\\
&= \left(\frac{3\cdot 4^2+1}{2\cdot 2}\right)^2 - 2\cdot 4 && \text{\# simplify in } \mathbb{F}_5\\
&= \left(\frac{3\cdot 1+1}{4}\right)^2 + 3\cdot 4 = \left(\frac{4}{4}\right)^2 + 2 = 1+2 = 3
\end{aligned}
$$

$$
\begin{aligned}
y' &= \left(\frac{3x^2+a}{2y}\right)^2 (x-x') - y && \text{\# insert points}\\
&= \left(\frac{3\cdot 4^2+1}{2\cdot 2}\right)^2 (4-3) - 2 && \text{\# simplify in } \mathbb{F}_5\\
&= 1\cdot 1 + 3 = 4
\end{aligned}
$$

So, in our elliptic curve $E_1(\mathbb{F}_5)$, we get the doubling of $(4,2)$, that is, $(4,2)\oplus(4,2) = (3,4)$, and, indeed the pair $(3,4)$ is an element of $E_1(\mathbb{F}_5)$ as expected. The group $E_1(\mathbb{F}_5)$ has no self-inverse points other than the neutral element $\mathcal{O}$, since no point has 0 as its y coordinate. We can invoke Sage to double-check the computations.

```
sage: F5 = GF(5)
sage: E1 = EllipticCurve(F5,[1,1])
sage: INF = E1(0) # point at infinity
sage: P1 = E1(0,1)
sage: P2 = E1(4,2)
sage: P3 = E1(0,4)
sage: R1 = E1(2,1)
sage: R2 = E1(3,4)
sage: R1 == P1+P2
True
sage: INF == P1+P3
True
sage: R2 == P2+P2
True
sage: R2 == 2*P2
True
sage: P3 == P3 + INF
True
```

Example 71 (Tiny-jubjub). Consider the *TJJ_13*-curve from example 66 again and recall that its group of rational points is given as follows: check reference

$$
\begin{aligned}
TJJ\_13 = \{&\mathcal{O},(1,2),(1,11),(4,0),(5,2),(5,11),(6,5),(6,8),(7,2),(7,11),\\
&(8,5),(8,8),(9,4),(9,9),(10,3),(10,10),(11,6),(11,7),(12,5),(12,8)\}
\end{aligned}
$$

In contrast to the group from the previous example, this group contains a self-inverse point, which is different from the neutral element, defined by $(4,0)$. To see what this means, observe that we cannot add $(4,0)$ to itself using the tangent rule 3 from definition 5.1.1.2, as the y coordinate is zero. Instead, we have to use rule 2, since $0 = -0$. We therefore get $(4,0)\oplus$ check reference

$(4,0) = \mathcal{O}$ in *TJJ_13*. The point $(4,0)$ is therefore the inverse of itself, as adding it to itself results in the neutral element.

```
sage: F13 = GF(13)
sage: MJJ = EllipticCurve(F13,[8,8])
sage: P = MJJ(4,0)
sage: INF = MJJ(0) # Point at infinity
sage: INF == P+P
True
sage: INF == 2*P
True
```

Example 72. Consider the Secp256k1 curve from example 67 again. The following code invokes Sage to generate a random affine curve point, then applies our compression method:

check reference

```
sage: P = Secp256k1.random_point()
sage: Q = Secp256k1.random_point()
sage: INF = Secp256k1(0)
sage: R1 = -P
sage: R2 = P + Q
sage: R3 = Secp256k1.order()*P
sage: P.xy()
(1119633893232454524081643097436326111729769989043529115485801
   76968926039239640, 1539003995352894081597153763406696148567
   2960162992773429658484259860711965820)
sage: Q.xy()
(5383721879850648652552441536168945917656985477576864548693858
   8859376837313427, 92227919932242341345636859214818301135596
   962711908163736135527905173637656195)
sage: (ZZ(R1[0]).str(16), ZZ(R1[1]).str(16))
('f789085b4bc1ac93df2743739024ea7c614d863f72b60523b830fb71b0e9
   ebd8', 'ddf98bcb7350e298a073ad531a0147d272f59bb0c153d75f018
   58a16f92077b3')
sage: R2.xy()
(7744488672817958943904552401263590302724076108809392748643668
   0826478595566401, 22311652849555765595410919852942556000795
   860353816350769477415065106850953095)
sage: R3 == INF
True
sage: P[1]+R1[1] == Fp(0) # -(x,y) = (x,-y)
True
```

Exercise 40. Consider the *TJJ_13*-curve from example 66.

check reference

1. Compute the inverse of $(10,10)$, $\mathcal{O}$, $(4,0)$ and $(1,2)$.
2. Compute the expression $3*(1,11)-(9,9)$.
3. Solve the equation $x+2(9,4) = (5,2)$ for some $x \in TJJ\_13$
4. Solve the equation $x \cdot (7,11) = (8,5)$ for $x \in \mathbb{Z}$

Scalar multiplication As we have seen in the previous section, elliptic curves $E(\mathbb{F})$ have the structure of a commutative group associated to them. Moreover, It can moreover be shown that this group is finite and cyclic whenever the field is finite.

To understand elliptic curve scalar multiplication, recall from page 44 that every finite cyclic group of order q has a generator g and an associated exponential map $g^{(\cdot)} : \mathbb{Z}_q \to \mathbb{G}$, where g^n is the n-fold product of g with itself.

check reference

Elliptic curve scalar multiplication is the exponential map written in additive notation. To be more precise, let $\mathbb{F}$ be a finite field, $E(\mathbb{F})$ an elliptic curve of order r, and P a generator of $E(\mathbb{F})$. Then the **elliptic curve scalar multiplication** with base P is defined as follows (where $[0]P = \mathcal{O}$ and $[m]P = P + P + \ldots + P$ is the m-fold sum of P with itself):

$$[\cdot]P : \mathbb{Z}_r \to E(\mathbb{F}); m \mapsto [m]P$$

therefore, elliptic curve scalar multiplication is an instantiation of the general exponential map using additive instead of multiplicative notation. This map is a homomorph of groups, which means that $[n+m]P = [n]P \oplus [m]P$.

As with all finite, cyclic groups, the inverse of the exponential map exists and is usually called the **elliptic curve discrete logarithm map**. However, elliptic curves are believed to be XXX-groups, which means that we don't know of any efficient way to actually compute this map.

add term

Scalar multiplication and its inverse, the elliptic curve discrete logarithm, define **the elliptic curve discrete logarithm problem**, which consists of finding solutions $m \in \mathbb{Z}_r$ such that the following equation holds:

$$P = [m]Q \tag{5.3}$$

Any solution m is usually called a **discrete logarithm relation** between P and Q. If Q is a generator of the curve, then there is a discrete logarithm relation between Q and any other point, since Q generates the group by repeatedly adding Q to itself. Therefore, we know that some discrete logarithm relation exists for generator Q and point P. However, since elliptic curves are believed to be XXX-groups, finding actual relations m is computationally hard, with runtimes being approximately the size of the order of the group. In practice, we often need the assumption that a discrete logarithm relation exists, while the relation itself is not known.

add term

One useful property of the exponential map in regard to the examples in this book is that it can be used to greatly simplify pen-and-paper computations. As we have seen in example XXX, computing the elliptic curve addition law takes quite a bit of effort when done without a computer. However, when g is a generator of a small pen-and-paper elliptic curve group of order r, we can use the exponential map to write the group using cofactor clearing, which implies that $[r]g = \mathcal{O}$:

add reference

cofactor clearing

$$\mathbb{G} = \{[1]g \to [2]g \to [3]g \to \cdots \to [r-1]g \to \mathcal{O}\} \tag{5.4}$$

"Logarithmic ordering" like this greatly simplifies complicated elliptic curve addition to the much simpler case of modular r addition. In order to add two curve points P and Q, we only have to look up their discrete log relations with the generator, say $P = [n]g$ and $Q = [m]g$, and compute the sum as $P \oplus Q = [n+m]g$. This is, of course, only possible for small groups where we can keep a clear overview, such as XXX.

add reference

In the following example, we will look at some implications of the fact that elliptic curves are finite cyclic groups. We will apply the fundamental theorem of finite cyclic groups and look how it reflects on the curves in consideration.

Example 73. Consider the elliptic curve group $E_1(\mathbb{F}_5)$ from example 65. Since it is a finite cyclic group of order 9, and the prime factorization of 9 is $3 \cdot 3$, we can use the fundamental

check reference

theorem of finite cyclic groups to reason about all its subgroups. In fact, since the only prime factor of 9 is 3, we know that $E_1(\mathbb{F}_5)$ has the following subgroups:

- $\mathbb{G}_1 = E_1(\mathbb{F}_5)$ is a subgroup of order 9. By definition, any group is a subgroup of itself.
- $\mathbb{G}_2 = \{(2,1),(2,4),\mathcal{O}\}$ is a subgroup of order 3. This is the subgroup associated to the prime factor 3.
- $\mathbb{G}_3 = \{\mathcal{O}\}$ is a subgroup of order 1. This is the trivial subgroup.

Moreover, since $E_1(\mathbb{F}_5)$ and all its subgroups are cyclic, we know from page 44 that they must have generators. For example, the curve point $(2,1)$ is a generator of the order 3 subgroup $\mathbb{G}_2$, since every element of $\mathbb{G}_2$ can be generated by repeatedly adding $(2,1)$ to itself: check reference

$$
\begin{aligned}
[1](2,1) &= (2,1)\\
[2](2,1) &= (2,4)\\
[3](2,1) &= \mathcal{O}
\end{aligned}
$$

Since $(2,1)$ is a generator, we know from XXX that it gives rise to an exponential map from the finite field $\mathbb{F}_3$ onto $\mathbb{G}_2$ defined by scalar multiplication: add reference

$$[\cdot](2,1) : \mathbb{F}_3 \to \mathbb{G}_2 \; : \; x \mapsto [x](2,1)$$

To give an example of a generator that generates the entire group $E_1(\mathbb{F}_5)$, consider the point $(0,1)$. Applying the tangent rule repeatedly, we compute as follows:

$$
\begin{array}{lcllcl}
[0](0,1) & = & \mathcal{O} & [1](0,1) & = & (0,1)\\
[2](0,1) & = & (4,2) & [3](0,1) & = & (2,1)\\
[4](0,1) & = & (3,4) & [5](0,1) & = & (3,1)\\
[6](0,1) & = & (2,4) & [7](0,1) & = & (4,3)\\
[8](0,1) & = & (0,4) & [9](0,1) & = & \mathcal{O}
\end{array}
$$

Again, since $(2,1)$ is a generator, we know from XXX that it gives rise to an exponential map. However, since the group order is not a prime number, the exponential map does not map from any field, but from the residue class ring $\mathbb{Z}_9$ only: add reference

$$[\cdot](0,1) : \mathbb{Z}_9 \to \mathbb{G}_1 \; : \; x \mapsto [x](0,1)$$

Using the generator $(0,1)$ and its associated exponential map, we can write $E(\mathbb{F}_1)$ i logarithmic order with respect to $(0,1)$ as explained in equation 5.4. We get the following: check reference

$$E_1(\mathbb{F}_5) = \{(0,1) \to (4,2) \to (2,1) \to (3,4) \to (3,1) \to (2,4) \to (4,3) \to (0,4) \to \mathcal{O}\}$$

This indicates that the first element is a generator, and the n-th element is the scalar product of n and the generator. To see how logarithmic orders like this simplify the computations in small elliptic curve groups, consider example 70 again. In that example, we use the chord-and-tangent rule to compute $(0,1) \oplus (4,2)$. Now, in the logarithmic order of $E_1(\mathbb{F})$, we can compute that sum much easier, since we can directly see that $(0,1) = [1](0,1)$ and $(4,2) = [2](0,1)$. We can then deduce $(0,1) \oplus (4,2) = (2,1)$ immediately, since $[1](0,1) \oplus [2](0,1) = [3](0,1) = (2,1)$. check reference

To give another example, we can immediately see that $(3,4) \oplus (4,3) = (4,2)$, without doing any expensive elliptic curve addition, since we know $(3,4) = [4](0,1)$ as well as $(4,3) =$

$[7](0,1)$ from the logarithmic representation of $E_1(\mathbb{F}_5)$. Since $4+7=2$ in $\mathbb{Z}_9$, the result must be $[2](0,1)=(4,2)$.

Finally we can use $E_1(\mathbb{F}_5)$ as an example to understand the concept of cofactor clearing from 5.4. Since the order of $E_1(\mathbb{F}_5)$ is 9, we only have a single factor, which happen to be the cofactor as well. Cofactor clearing then implies that we can map any element from $E_1(\mathbb{F}_5)$ onto its prime factor group $\mathbb{G}_2$ by scalar multiplication with 3. For example, taking the element $(3,4)$, which is not in $\mathbb{G}_2$, and multiplying it with 3, we get $[3](3,4)=(2,1)$, which is an element of $\mathbb{G}_2$ as expected.

check reference

In the following example, we will look at the subgroups of our tiny-jubjub curve, define generators, and compute the logarithmic order for pen-and-paper computations. Then we take another look at the principle of cofactor clearing.

Example 74. Consider the tiny-jubjub curve *TJJ_13* from example 66 again. Since the order of *TJJ_13* is 20, and the prime factorization of 20 is $2^2\cdot 5$, we know that the *TJJ_13* contains a "large" prime-order subgroup of size 5 and a small prime oder subgroup of size 2.

check reference

To compute those groups, we can apply the technique of cofactor clearing in a try-and-repeat loop. We start the loop by arbitrarily choosing an element $P\in$ *TJJ_13*, then multiplying that element with the cofactor of the group that we want to compute. If the result is $\mathcal{O}$, we try a different element and repeat the process until the result is different from the point at infinity $\mathcal{O}$.

To compute a generator for the small prime-order subgroup $(TJJ\_13)_2$, first observe that the cofactor is 10, since $20=2\cdot 10$. We then arbitrarily choose the curve point $(5,11)\in$ *TJJ_13* and compute $[10](5,11)=\mathcal{O}$. Since the result is the point at infinity, we have to try another curve point, say $(9,4)$. We get $[10](9,4)=(4,0)$ and we can deduce that $(4,0)$ is a generator of $(TJJ\_13)_2$. Logarithmic order then gives $(TJJ\_13)_2=\{(4,0)\to\mathcal{O}\}$ as expected, since we know from example 71 that $(4,0)$ is self-inverse, with $(4,0)\oplus(4,0)=\mathcal{O}$. We double check the computations using Sage:

check reference

```
sage: F13 = GF(13)
sage: TJJ = EllipticCurve(F13,[8,8])
sage: P = TJJ(5,11)
sage: INF = TJJ(0)
sage: 10*P == INF
True
sage: Q = TJJ(9,4)
sage: R = TJJ(4,0)
sage: 10*Q == R
True
```

We can apply the same reasoning to the "large" prime-order subgroup $(TJJ\_13)_5$, which contains 5 elements. To compute a generator for this group, first observe that the associated cofactor is 4, since $20=5\cdot 4$. We choose the curve point $(9,4)\in$ *TJJ_13* again, and compute $[4](9,4)=(7,11)$. We can deduce that $(7,11)$ is a generator of $(TJJ\_13)_5$. Using the generator $(7,11)$, we compute the exponential map $[\cdot](7,11):\mathbb{F}_5\to$ *TJJ_13* and get the following:

Explain how

$$
\begin{aligned}
[0](7,11)&=\mathcal{O}\\
[1](7,11)&=(7,11)\\
[2](7,11)&=(8,5)\\
[3](7,11)&=(8,8)\\
[4](7,11)&=(7,2)
\end{aligned}
$$

We can use this computation to write the large-order prime group $(TJJ\_13)_5$ of the tiny-jubjub curve in logarithmic order, which we will use quite frequently in what follows. We get the following:

$$(TJJ\_13)_5 = \{(7,11) \to (8,5) \to (8,8) \to (7,2) \to \mathcal{O}\} \tag{5.5}$$

From this, we can immediately see, for example that $(8,8) \oplus (7,2) = (8,5)$, since $3+4=2$ in $\mathbb{F}_5$.

From the previous two examples, the reader might get the impression that elliptic curve computation can be largely replaced by modular arithmetics. This however, is not true in general, but only an artifact of small groups, where it is possible to write the entire group in a logarithmic order. The following example gives some understanding of why this is not possible in cryptographically secure groups.

Example 75. SEKTP BICOIN. DISCRETE LOG HARDNESS PROHIBITS ADDITION IN THE FIELD...

write example

Projective short Weierstraß form As we have seen in the previous section, describing elliptic curves as pairs of points that satisfy a certain equation is relatively straight-forward. However, in order to define a group structure on the set of points, we had to add a special point at infinity to act as the neutral element.

Recalling from the definition of projective planes (section 4.4), we know that points at infinity are handled as ordinary points in projective geometry. Therefore, it makes sense to look at the definition of a short Weierstraß curve in projective geometry.

check reference

To see what a short Weierstraß curve in projective coordinates is, let $\mathbb{F}$ be a finite field of order q and characteristic > 3, let $a,b \in \mathbb{F}$ be two field elements such that $4a^3 + 27b^2 \bmod q \neq 0$ and let $\mathbb{F}\mathrm{P}^2$ be the projective plane over $\mathbb{F}$. Then a **short Weierstraß elliptic curve** over $\mathbb{F}$ in its projective representation is the set of all points $[X:Y:Z] \in \mathbb{F}\mathrm{P}^2$ from the projective plane that satisfy the **homogenous** cubic equation $Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3$:

$$E(\mathbb{F}\mathrm{P}^2) = \{[X:Y:Z] \in \mathbb{F}\mathrm{P}^2 \mid Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3\} \tag{5.6}$$

To understand how the point at infinity is unified in this definition, recall from XXX that, in projective geometry, points at infinity are given by homogeneous coordinates $[X:Y:0]$. Inserting representatives $(x_1, y_1, 0) \in [X:Y:0]$ from those classes into the defining homogenous cubic equations gives the following:

add reference

$$\begin{aligned} y_1^2 \cdot 0 &= x_1^3 + a \cdot x_1 \cdot 0^2 + b \cdot 0^3 && \Leftrightarrow \\ 0 &= x_1^3 \end{aligned}$$

This shows that the only point at infinity, that is also a point on a projective short Weierstraß curve is the class $[0,1,0] = \{(0,y,0) \mid y \in \mathbb{F}\}$.

This point is the projective representation of $\mathcal{O}$. The projective representation of a short Weierstraß curve, therefore, has the advantage that it does not need a special symbol to represent the point at infinity $\mathcal{O}$ from the affine definition.

Example 76. To get an intuition of how an elliptic curve in projective geometry looks, consider curve $E_1(\mathbb{F}_5)$ from example (65). We know that, in its affine representation, the set of rational points is given as follows:

check reference

$$E_1(\mathbb{F}_5) = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\} \tag{5.7}$$

This is defined as the set of all pairs $(x,y) \in \mathbb{F}_5 \times \mathbb{F}_5$ such that the affine short Weierstraß equation $y^2 = x^3 + ax + b$ with $a = 1$ and $b = 1$ is satisfied.

To find the projective representation of a short Weierstraß curve with the same parameters $a = 1$ and $b = 1$, we have to compute the set of projective points $[X : Y : Z]$ from the projective plane $\mathbb{F}_5\mathrm{P}^2$ that satisfy the following homogenous cubic equation for any representative $(x_1, y_1, z_1) \in [X : Y : Z]$:

$$y_1^2 z_1 = x_1^3 + 1 \cdot x_1 z_1^2 + 1 \cdot z_1^3 \tag{5.8}$$

We know from XXX that the projective plane $\mathbb{F}_5\mathrm{P}^2$ contains $5^2 + 5 + 1 = 31$ elements, so we can take the effort and insert all elements into equation 5.8 and see if both sides match.

add reference

For example, consider the projective point $[0 : 4 : 1]$. We know from XXX that this point in the projective plane represents the following line in the three-dimensional space $\mathbb{F}^3$:

check reference

add reference

$$[0 : 4 : 1] = \{(0,0,0), (0,4,1), (0,3,2), (0,2,3), (0,1,4)\}$$

To check whether or not $[0 : 4 : 1]$ satisfies 5.8, we can insert any representative, in other words, any element from XXX. Each element satisfies the equation if and only if all other elements satisfy the equation. So, we insert $(0,4,1)$ and get the following result:

check reference

add reference

$$1^2 \cdot 1 = 0^3 + 1 \cdot 0 \cdot 1^2 + 1 \cdot 1^3$$

This tells us that the affine point $[0 : 4 : 1]$ is indeed a solution to the equation 5.8, but we could just as well have inserted any other representative. For example, inserting $(0,3,2)$ also satisfies 5.8:

check reference

$$3^2 \cdot 2 = 0^3 + 1 \cdot 0 \cdot 2^2 + 1 \cdot 2^3$$

To find the projective representation of E_1, we first observe that the projective line at infinity $[1 : 0 : 0]$ is not a curve point on any projective short Weierstraß curve, since it cannot satisfy XXX for any parameter a and b. Therefore, we can exclude it from our consideration.

add reference

Moreover, a point at infinity $[X : Y : 0]$ can only satisfy equation XXX for any a and b, if $X = 0$, which implies that the only point at infinity relevant for short Weierstraß elliptic curves is $[0 : 1 : 0]$, since $[0 : k : 0] = [0 : 1 : 0]$ for all k from the finite field. Therefore, we can exclude all points at infinity except the point $[0 : 1 : 0]$.

add reference

All points that remain are the affine points $[X : Y : 1]$. Inserting all of them into XXX, we get the set of all projective curve points as follows:

add reference

$$\begin{aligned} E_1(\mathbb{F}_5\mathrm{P}^2) = \{&[0:1:0], [0:1:1], [2:1:1], [3:1:1], \\ &[4:2:1], [4:3:1], [0:4:1], [2:4:1], [3:4:1]\} \end{aligned}$$

If we compare this with the affine representation, we see that there is a 1:1 correspondence between the points in the affine representation in 5.7 and the affine points in projective geometry, and that the point $[0 : 1 : 0]$ represents the additional point $\mathcal{O}$ in the projective representation.

check reference

Exercise 41. Compute the projective representation of the tiny-jubjub curve and the logarithmic order of its large prime-order subgroup with respect to the generator $(7,11)$.

Projective Group law As we have seen on page 71, one of the key properties of an elliptic curve is that it comes with a definition of a group law on the set of its rational points, described geometrically by the chord-and-tangent rule (definition 5.1.1.1). This rule was kind of intuitive,

check reference

with the exception of the distinguished point at infinity, which appeared whenever the chord or the tangent did not have a third intersection point with the curve.

One of the key features of projective coordinates is that, in projective space, it is guaranteed that any chord will always intersect the curve in three points, and any tangent will intersect it in two points including the tangent point. So, the geometric picture simplifies, as we don't need to consider external symbols and associated cases.

Again, it can be shown that the points of an elliptic curve in projective space form a commutative group with respect to the tangent-and-chord rule such that the projective point $[0:1:0]$ is the neutral element, and the additive inverse of a point $[X:Y:Z]$ is given by $[X:-Y:Z]$. The addition law is usually described by the following algorithm, minimizing the number of necessary additions and multiplications in the base field.

Check if following Alg is floated too far

Exercise 42. Compare the affine addition law for short Weierstraß curves with the projective addition rule. Which branch in the projective rule corresponds to which case in the affine law?

Coordinate Transformations As we have seen in example XXX, there was a close relation between the affine and the projective representation of a short Weierstraß curve. This was not a coincidence. In fact, from a mathematical point of view, projective and affine short Weierstraß curves describe the same thing, as there is a one-to-one correspondence (an isomorphism) between both representations for any arbitrary parameters a and b.

add reference

To specify the isomorphism, let $E(\mathbb{F})$ and $E(\mathbb{F}\mathrm{P}^2)$ be an affine and a projective short Weierstraß curve defined for the same parameters a and b. Then the map in 5.9 maps points from the affine representation to points from the projective representation of a short Weierstraß curve. In other words, if the pair of points (x,y) satisfies the affine equation $y^2 = x^3+ax+b$, then all homogeneous coordinates $(x_1,y_1,z_1) \in [x:y:1]$ satisfy the projective equation $y_1^2\cdot z_1 = x_1^3+ay_1\cdot z_1^2+b\cdot z_1^3$.

$$\Phi : E(\mathbb{F}) \to E(\mathbb{F}\mathrm{P}^2) \; : \; \begin{array}{ccc} (x,y) & \mapsto & [x:y:1] \\ \mathcal{O} & \mapsto & [0:1:0] \end{array} \tag{5.9}$$

The inverse is given by the following map:

$$\Phi^{-1} : E(\mathbb{F}\mathbb{P}^2) \to E(\mathbb{F}) \; : \; [X:Y:Z] \mapsto \begin{cases} (\frac{X}{Z},\frac{Y}{Z}) & \text{if } Z \neq 0 \\ \mathcal{O} & \text{if } Z = 0 \end{cases} \tag{5.10}$$

Note that the only projective point $[X:Y:Z]$ with $Z \neq 0$ that satisfies XXX is given by the class $[0:1:0]$.

add reference

One key feature of Φ and its inverse is that it respects the group structure, which means that $\Phi((x_1,y_1)\oplus(x_2,y_2))$ is equal to $\Phi(x_1,y_1)\oplus\Phi(x_2,y_2)$. The same holds true for the inverse map Φ^{-1}.

Maps with these properties are called **group isomorphisms**, and, from a mathematical point of view, the existence of Φ implies that these two definitions are equivalent, and implementations can choose freely between these representations.

5.1.2 Montgomery Curves

History and use of them (optimized scalar multiplication)

write up this part

Algorithm 6 Projective Weierstraß Addition Law

Require: $[X_1 : Y_1 : Z_1], [X_2 : Y_2 : Z_2] \in E(\mathbb{FP}^2)$
```
procedure ADD-RULE([X1 : Y1 : Z1], [X2 : Y2 : Z2])
    if [X1 : Y1 : Z1] == [0 : 1 : 0] then
        [X3 : Y3 : Z3] ← [X2 : Y2 : Z2]
    else if [X2 : Y2 : Z2] == [0 : 1 : 0] then
        [X3 : Y3 : Z3] ← [X1 : Y1 : Z1]
    else
        U1 ← Y2 · Z1
        U2 ← Y1 · Z2
        V1 ← X2 · Z1
        V2 ← X1 · Z2
        if V1 == V2 then
            if U1 ≠ U2 then [X3 : Y3 : Z3] ← [0 : 1 : 0]
            else
                if Y1 == 0 then [X3 : Y3 : Z3] ← [0 : 1 : 0]
                else
                    W ← a · Z1^2 + 3 · X1^2
                    S ← Y1 · Z1
                    B ← X1 · Y1 · S
                    H ← W^2 − 8 · B
                    X' ← 2 · H · S
                    Y' ← W · (4 · B − H) − 8 · Y1^2 · S^2
                    Z' ← 8 · S^3
                    [X3 : Y3 : Z3] ← [X' : Y' : Z']
                end if
            end if
        else
            U = U1 − U2
            V = V1 − V2
            W = Z1 · Z2
            A = U^2 · W − V^3 − 2 · V^2 · V2
            X' = V · A
            Y' = U · (V^2 · V2 − A) − V^3 · U2
            Z' = V^3 · W
            [X3 : Y3 : Z3] ← [X' : Y' : Z']
        end if
    end if
    return [X3 : Y3 : Z3]
end procedure
```
Ensure: $[X_3 : Y_3 : Z_3] == [X_1 : Y_1 : Z_1] \oplus [X_2 : Y_2 : Z_2]$

Affine Montgomery Form To see what a Montgomery curve in affine coordinates is, let $\mathbb{F}$ be a finite field of characteristic > 2, and let $A, B \in \mathbb{F}$ be two field elements such that $B \neq 0$ and $A^2 \neq 4$. A **Montgomery elliptic curve** $M(\mathbb{F})$ over $\mathbb{F}$ in its affine representation is the set of all pairs of field elements $(x,y) \in \mathbb{F} \times \mathbb{F}$ that satisfy the Montgomery cubic equation $B \cdot y^2 = x^3 + A \cdot x^2 + x$, together with a distinguished symbol $\mathcal{O}$, called the **point at infinity**.

$$M(\mathbb{F}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid B \cdot y^2 = x^3 + A \cdot x^2 + x\} \bigcup \{\mathcal{O}\} \tag{5.11}$$

is the label in LaTeX correct here?

Despite the fact that Montgomery curves look different from short Weierstraß curves, they are just a special way to describe certain short Weierstraß curves. In fact, every curve in affine Montgomery form can be transformed into an elliptic curve in Weierstraß form. To see that, assume that a curve is given in Montgomery form $By^2 = x^3 + Ax^2 + x$. The associated Weierstraß form is then as follows:

$$y^2 = x^3 + \frac{3 - A^2}{3B^2} \cdot x + \frac{2A^3 - 9A}{27B^3} \tag{5.12}$$

On the other hand, an elliptic curve $E(\mathbb{F})$ over base field $\mathbb{F}$ in Weierstraß form $y^2 = x^3 + ax + b$ can be converted to Montgomery form if and only if the following conditions hold:

Definition 5.1.2.1. **Requirements for Montgomery curves**

- The number of points on $E(F)$ is divisible by 4
- The polynomial $z^3 + az + b \in \mathbb{F}[z]$ has at least one root $z_0 \in \mathbb{F}$
- $3z_0^2 + a$ is a quadratic residue in $\mathbb{F}$.

When these conditions are satisfied, then for $s = (\sqrt{3z_0^2 + a})^{-1}$, the equivalent Montgomery curve is defined by the following equation:

$$sy^2 = x^3 + (3z_0 s)x^2 + x \tag{5.13}$$

In the following example we will look at our tiny-jubjub curve again, and show that it is actually a Montgomery curve.

Example 77. Consider the prime field $\mathbb{F}_{13}$ and the tiny-jubjub curve *TJJ_13* from example 66. To see that it is a Montgomery curve, we have to check the requirements from 5.1.2.1: check reference

check reference

Since the order of *TJJ_13* is 20, which is divisible by 4, the first requirement is met.

Next, since $a = 8$ and $b = 8$, we have to check if the polynomial $P(z) = z^3 + 8z + 8$ has a root in $\mathbb{F}_{13}$. We simply evaluate P at all numbers $z \in \mathbb{F}_{13}$, and find that $P(4) = 0$, so a root is given by $z_0 = 4$.

In the last step, we have to check that $3 \cdot z_0^2 + a$ has a root in $\mathbb{F}_{13}$. We compute as follows:

$$\begin{aligned} 3z_0^2 + a &= 3 \cdot 4^2 + 8 \\ &= 3 \cdot 3 + 8 \\ &= 9 + 8 \\ &= 4 \end{aligned}$$

To see if 4 is a quadratic residue, we can use Euler's criterion (4.28) to compute the Legendre symbol of 4. We get the following: check reference

$$\left(\frac{4}{13}\right) = 4^{\frac{13-1}{2}} = 4^6 = 1$$

This means that 4 does have a root in $\mathbb{F}_{13}$. In fact, computing a root of 4 in $\mathbb{F}_{13}$ is easy, since the integer root 2 of 4 is also one of its roots in $\mathbb{F}_{13}$. The other root is given by $13-4=9$.

Since all requirements are meet, we have now shown that *TJJ_13* is indeed a Montgomery curve, and we can use 5.13 to compute its associated Montgomery form. We compute as follows:

check reference

$$\begin{aligned}
s &= \left(\sqrt{3\cdot z_0^2+8}\right)^{-1} \\
&= 2^{-1} && \text{\# Fermat's little theorem} \\
&= 2^{13-2} && \text{\# 2048 mod 13} = 7 \\
&= 7
\end{aligned}$$

The defining equation for the Montgomery form of our tiny-jubjub curve is then given by the following equation:

$$\begin{aligned}
sy^2 &= x^3 + (3z_0s)x^2 + x && \Rightarrow \\
7\cdot y^2 &= x^3 + (3\cdot 4\cdot 7)x^2 + x && \Leftrightarrow \\
7\cdot y^2 &= x^3 + 6x^2 + x
\end{aligned}$$

So, we get the defining parameters as $B=7$ and $A=6$, and we can write the tiny-jubjub curve in its affine Montgomery representation as follows:

$$TJJ\_13 = \{(x,y)\in\mathbb{F}_{13}\times\mathbb{F}_{13} \mid 7\cdot y^2 = x^3+6x^2+x\}\bigcup\{\mathcal{O}\} \tag{5.14}$$

Now that we have the abstract definition of our tiny-jubjub curve in Montgomery form, we can compute the set of points by inserting all pairs $(x,y)\in\mathbb{F}_{13}\times\mathbb{F}_{13}$ similarly to how we computed the curve points in its Weierstraß representation. We get the following:

$$\begin{aligned}
TJJ_13 = \{&\mathcal{O},(0,0),(1,4),(1,9),(2,4),(2,9),(3,5),(3,8),(4,4),(4,9),\\
&(5,1),(5,12),(7,1),(7,12),(8,1),(8,12),(9,2),(9,11),(10,3),(10,10)\}
\end{aligned}$$

```
sage: F13 = GF(13)
sage: L_MTJJ = []
....: for x in F13:
....:     for y in F13:
....:         if F13(7)*y^2 == x^3 + F13(6)*x^2 +x:
....:             L_MTJJ.append((x,y))
sage: MTJJ = Set(L_MTJJ)
sage: # does not compute the point at infinity
```

Affine Montgomery coordinate transformation Comparing the Montgomery representation of the previous example (equation 5.14) with the Weierstraß representation of the same curve (equation 5.2), we see that there is a 1:1 correspondence between the curve points in both

check reference

examples. This is no accident. In fact, if $M_{A,B}$ is a Montgomery curve, and $E_{a,b}$ a Weierstraß curve with $a = \frac{3-A^2}{3B^2}$ and $b = \frac{2A^2-9A}{27B^3}$ then the following function maps all points in Montgomery representation onto the points in Weierstraß representation:

$$\Phi : M_{A,B} \to E_{a,b} \;:\; (x,y) \mapsto \left(\frac{3x+A}{3B}, \frac{y}{B}\right) \tag{5.15}$$

This map is a 1:1 correspondence (am isomorphism), and its inverse map is given by the following equation (where z_0 is a root of the polynomial $z^3+az+b \in \mathbb{F}[z]$ and $s = (\sqrt{3z_0^2+a})^{-1}$).

$$\Phi^{-1} : E_{a,b} \to M_{A,B} \;:\; (x,y) \mapsto (s\cdot(x-z_0), s\cdot y) \tag{5.16}$$

Using this map, it is therefore possible for implementations of Montgomery curves to freely transit between the Weierstraß and the Montgomery representation. However, as we saw in definition 5.1.2.1, not every Weierstraß curve is a Montgomery curve, as all criteria in 5.1.2.1 have to be satisfied. This means that the map Φ^{-1} does not always exist.

check reference

Example 78. Consider our tiny-jubjub curve again. In equation 5.2 we derived its Weierstraß representation and in example 5.14, we derived its Montgomery representation.

check reference

To see how coordinate transformation Φ works in this example, let's map points from the Montgomery representation onto points from the Weierstraß representation. Inserting, for example, the point $(0,0)$ from the Montgomery representation 5.14 into Φ gives the following:

check reference

check reference

check reference

$$\begin{aligned}
\Phi(0,0) &= \left(\frac{3\cdot 0+A}{3B}, \frac{0}{B}\right)\\
&= \left(\frac{3\cdot 0+6}{3\cdot 7}, \frac{0}{7}\right)\\
&= \left(\frac{6}{8}, 0\right)\\
&= (4,0)
\end{aligned}$$

As we can see, the Montgomery point $(0,0)$ maps to the self-inverse point $(4,0)$ of the Weierstraß representation. On the other hand, we can use our computations of $s = 7$ and $z_0 = 4$ from XXX to compute the inverse map Φ^{-1}, which maps points on the Weiertraß representation to points on the Mongomery form. Inserting, for example, $(4,0)$ we get the following:

add reference

$$\begin{aligned}
\Phi^{-1}(4,0) &= (s\cdot(4-z_0), s\cdot 0)\\
&= (7\cdot(4-4), 0)\\
&= (0,0)
\end{aligned}$$

As expected, the inverse map maps the Weierstraß point back to where it originated in the Montgomery form. We can invoke Sage to check that our computation of Φ is correct:

```
sage: # Compute PHI of Montgomery form:
sage: L_PHI_MTJJ = []
sage: for (x,y) in L_MTJJ: # LMJJ as defined previously
....:     v = (F13(3)*x + F13(6))/(F13(3)*F13(7))
....:     w = y/F13(7)
....:     L_PHI_MTJJ.append((v,w))
```

```
sage: PHI_MTJJ = Set(L_PHI_MTJJ)
sage: # Computation Weierstrass form
sage: C_WTJJ = EllipticCurve(F13,[8,8])
sage: L_WTJJ = [P.xy() for P in C_WTJJ.points() if P.order() >
    1]
sage: WTJJ = Set(L_WTJJ)
sage: # check PHI(Montgomery) == Weierstrass
sage: WTJJ == PHI_MTJJ
True
sage: # check the inverse map PHI^(-1)
sage: L_PHIINV_WTJJ = []
sage: for (v,w) in L_WTJJ:
....:     x = F13(7)*(v-F13(4))
....:     y = F13(7)*w
....:     L_PHIINV_WTJJ.append((x,y))
sage: PHIINV_WTJJ = Set(L_PHIINV_WTJJ)
sage: MTJJ == PHIINV_WTJJ
True
```

Montgomery group law We have seen that Montgomery curves special cases of short Weierstraß curves. As such, they have a group structure defined on the set of their points, which can also be derived from the chord-and-tangent rule. In accordance with short Weierstraß curves, it can be shown that the identity $x_1 = x_2$ implies $y_2 = \pm y_1$, meaning that the following rules are a complete description of the affine addition law.

Definition 5.1.2.2. **Montgomery group law**

- (Neutral element) Point at infinity $\mathcal{O}$ is the neutral element.
- (Additive inverse) The additive inverse of $\mathcal{O}$ is $\mathcal{O}$. For any other curve point $(x,y) \in M(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, the additive inverse is given by $(x,-y)$.
- (Addition rule) For any two curve points $P, Q \in M(\mathbb{F}_q)$, addition is defined by one of the following cases:
    1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P + Q = P$.
    2. (Adding inverse elements) If $P = (x,y)$ and $Q = (x,-y)$, the sum is defined as $P + Q = \mathcal{O}$.
    3. (Adding non-self-inverse equal points) If $P = (x,y)$ and $Q = (x,y)$ with $y \neq 0$, the sum $2P = (x',y')$ is defined as follows:

    $$x' = \left(\frac{3x_1^2+2Ax_1+1}{2By_1}\right)^2 \cdot B - (x_1+x_2) - A \quad , \quad y' = \frac{3x_1^2+2Ax_1+1}{2By_1}(x_1 - x') - y_1$$

    4. (Adding non-inverse different points) If $P = (x_1,y_1)$ and $Q = (x_2,y_2)$ such that $x_1 \neq x_2$, the sum $R = P + Q$ with $R = (x_3,y_3)$ is defined as follows:

    $$x' = \left(\frac{y_2-y_1}{x_2-x_1}\right)^2 B - (x_1+x_2) - A \quad , \quad y' = \frac{y_2-y_1}{x_2-x_1}(x_1 - x') - y_1$$

5.1.3 Twisted Edwards Curves

As we have seen in 5.1.2.2 both Weierstraß and Montgomery curves have somewhat complicated addition and doubling laws, as many cases have to be distinguished. Those various cases translate to branches in computer programs.

check reference

In the context of SNARK development, two computational models for bounded computations are used, called **circuits** and **rank-1 constraint systems**. Program branches are undesirably costly when implemented in those models. It is therefore advantageous to look for curves with an addition/doubling rule that requires no branches and as few field operations as possible.

Twisted Edwards curves are particularly useful here, as a subclass of these curves has a compact and easily implementable addition law that works for all points including the point at infinity. Implementing this law needs no branching.

Twisted Edwards Form To see what an affine **twisted Edwards curve** looks like, let $\mathbb{F}$ be a finite field of characteristic > 2, and let $a, d \in \mathbb{F}\backslash\{0\}$ be two non-zero field elements with $a \neq d$. A **twisted Edwards elliptic curve** in its affine representation is the set of all pairs (x, y) from $\mathbb{F} \times \mathbb{F}$ that satisfy the twisted Edwards equation $a \cdot x^2 + y^2 = 1 + d \cdot x^2y^2$, given below:

$$E(\mathbb{F}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid a \cdot x^2 + y^2 = 1 + d \cdot x^2y^2\} \tag{5.17}$$

A twisted Edwards curve is called an **Edwards curve (non-twisted)**, if the parameter a is equal to 1, and it is called a **SNARK-friendly twisted Edwards curve** if the parameter a is a quadratic residue and the parameter d is a quadratic non-residue.

As we can see from the definition, affine twisted Edwards curves look somewhat different from Weierstraß curves, as their affine representation does not need a special symbol to represent the point at infinity. In fact, we we will see that the pair $(0, 1)$ is always a point on any twisted Edwards curve, and that it takes the role of the point at infinity.

Despite their different appearances however, twisted Edwards curves are equivalent to Montgomery curves in the sense that, for every twisted Edwards curve, there is a Montgomery curve, and a way to map the points of one curve in a 1:1 correspondence onto the other and vice versa. To see that, assume that a curve in twisted Edwards form $a \cdot x^2 + y^2 = 1 + d \cdot x^2y^2$ is given. The associated Montgomery curve is then defined by the Montgomery equation:

$$\frac{4}{a-d}y^2 = x^3 + \frac{2(a+d)}{a-d} \cdot x^2 + x \tag{5.18}$$

On the other hand, a Montgomery curve $By^2 = x^3 + Ax^2 + x$ with $B \neq 0$ and $A^2 \neq 4$ can give rise to a twisted Edwards curve defined by the following equation:

$$(\frac{A+2}{B})x^2 + y^2 = 1 + (\frac{A-2}{B})x^2y^2 \tag{5.19}$$

As we have seen in equation 5.12 and the following discussion, Montgomery curves are just a special class of Weierstraß curves. Furthermore we now know that twisted Edwards curves are special Weierstraß curves too. This means that the more general way to describe elliptic curves is as Weierstraß curves.

check reference

Example 79. Consider the tiny-jubjub curve from example 66 again. We know from example 77 that it is a Montgomery curve, and, since Montgomery curves are equivalent to twisted Edwards curves, we want to write this curve in twisted Edwards form. We use equation 5.19,

check reference

check reference

check reference

and compute the parameters a and d as follows:

$$\begin{aligned} a &= \frac{A+2}{B} && \text{\# insert A=6 and B=7} \\ &= \frac{8}{7} = 3 && \text{\# } 7^{-1} = 2 \end{aligned}$$

$$\begin{aligned} d &= \frac{A-2}{B} \\ &= \frac{4}{7} = 8 \end{aligned}$$

Thus, we get the defining parameters as $a = 3$ and $d = 8$. Since our goal is to use this curve later on in implementations of pen-and-paper SNARKs, let us show that tiny-jubjub is also a **SNARK-friendly** twisted Edwards curve. To see that, we have to show that a is a quadratic residue and d is a quadratic non-residue. We therefore compute the Legendre symbols of a and d using Euler's criterion. We get the following:

$$\begin{aligned} \left(\frac{3}{13}\right) &= 3^{\frac{13-1}{2}} \\ &= 3^6 = 1 \end{aligned}$$

$$\begin{aligned} \left(\frac{8}{13}\right) &= 8^{\frac{13-1}{2}} \\ &= 8^6 = 12 = -1 \end{aligned}$$

This proves that tiny-jubjub is SNARK-friendly. We can write the tiny-jubjub curve in its affine twisted Edwards representation as follows:

$$TJJ\_13 = \{(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13} \mid 3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2\} \tag{5.20}$$

Now that we have the abstract definition of our tiny-jubjub curve in twisted Edwards form, we can compute the set of points by inserting all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$, similarly to how we computed the curve points in its Weierstraß or Edwards representation. We get the following:

$$\begin{aligned} TJJ\_13 = \{&(0,1),(0,12),(1,2),(1,11),(2,6),(2,7),(3,0),(5,5),(5,8),(6,4), \\ &(6,9),(7,4),(7,9),(8,5),(8,8),(10,0),(11,6),(11,7),(12,2),(12,11)\} \end{aligned} \tag{5.21}$$

```
sage: F13 = GF(13)
sage: L_ETJJ = []
....: for x in F13:
....:     for y in F13:
....:         if F13(3)*x^2 + y^2 == 1+ F13(8)*x^2*y^2:
....:             L_ETJJ.append((x,y))
sage: ETJJ = Set(L_ETJJ)
```

Twisted Edwards group law As we have seen, twisted Edwards curves are equivalent to Montgomery curves, and, as such, also have a group law. However, in contrast to Montgomery and Weierstraß curves, the group law of SNARK-friendly twisted Edwards curves can be described by a single computation that works in all cases, no matter if we add the neutral element, the inverse, or if we have to double a point. To see what the group law looks like, first observe that the point $(0,1)$ is a solution to $a\cdot x^2+y^2=1+d\cdot x^2\cdot y^2$ for any curve. The sum of any two points (x_1,y_1), (x_2,y_2) on an Edwards curve $E(\mathbb{F})$ is then given by the following equation:

$$(x_1,y_1)\oplus(x_2,y_2)=\left(\frac{x_1y_2+y_1x_2}{1+dx_1x_2y_1y_2},\frac{y_1y_2-ax_1x_2}{1-dx_1x_2y_1y_2}\right) \tag{5.22}$$

and it can be shown that the point $(0,1)$ serves as the neutral element and the inverse of a point (x_1,y_1) is given by $(-x_1,y1)$.

Example 80. Lets look at the tiny-jubjub curve in Edwards form from example 5.20 again. As we have seen, this curve is given by check reference

$$\begin{aligned}TJJ\_13=\{&(0,1),(0,12),(1,2),(1,11),(2,6),(2,7),(3,0),(5,5),(5,8),(6,4),\\&(6,9),(7,4),(7,9),(8,5),(8,8),(10,0),(11,6),(11,7),(12,2),(12,11)\}\end{aligned}$$

To get an understanding of the twisted Edwards addition law, let's first add the neutral element $(0,1)$ to itself. We apply the group law 5.22 and get the following: check reference

$$\begin{aligned}(0,1)\oplus(0,1)&=\left(\frac{0\cdot1+1\cdot0}{1+8\cdot0\cdot0\cdot1\cdot1},\frac{1\cdot1-3\cdot0\cdot0}{1-8\cdot0\cdot0\cdot1\cdot1}\right)\\&=(0,1)\end{aligned}$$

So, as expected, the neutral element added to itself gives the neutral element again. Now let's add the neutral element to some other curve point. We get the following:

$$\begin{aligned}(0,1)\oplus(8,5)&=\left(\frac{0\cdot5+1\cdot8}{1+8\cdot0\cdot8\cdot1\cdot5},\frac{1\cdot5-3\cdot0\cdot8}{1-8\cdot0\cdot8\cdot1\cdot5}\right)\\&=(8,5)\end{aligned}$$

Again, as expected, adding the neutral element to any element will result in that element again. Given any curve point (x,y), we know that its inverse is given by $(-x,y)$. To see how the addition of a point to its inverse works, we compute as follows:

$$\begin{aligned}(5,5)\oplus(8,5)&=\left(\frac{5\cdot5+5\cdot8}{1+8\cdot5\cdot8\cdot5\cdot5},\frac{5\cdot5-3\cdot5\cdot8}{1-8\cdot5\cdot8\cdot5\cdot5}\right)\\&=\left(\frac{12+1}{1+5},\frac{12-3}{1-5}\right)\\&=\left(\frac{0}{6},\frac{12+10}{1+8}\right)\\&=\left(0,\frac{9}{9}\right)\\&=(0,1)\end{aligned}$$

Adding a curve point to its inverse gives the neutral element, as expected. As we have seen from these examples, the twisted Edwards addition law handles edge cases particularly well and in a unified way.

5.2 Elliptic Curve Pairings

As we have seen in equation 4.1, some groups come with the notation of a so-called pairing map, which is a non-degenerate bilinear map from two groups into another group. check reference

In this section, we discuss **pairings on elliptic curves**, which form the basis of several zk-SNARKs and other zero-knowledge proof schemes. The SNARKs derived from pairings have the advantage of constant proof sizes, which is crucial to blockchains.

We start out by defining elliptic curve pairings and discussing a simple application which bears some resemblance to more advanced SNARKs. We then introduce the pairings arising from elliptic curves and describe Miller's algorithm, which makes these pairings practical rather than just theoretically interesting.

Elliptic curves have a few structures, like the Weil or the Tate map that qualifies as pairing. either expand on this or delete it

Embedding Degrees As we will see in what follows, every elliptic curves gives rise to a pairing map. However, we will also see in example XXX that not every such pairing can be efficiently computed. In order to distinguish curves with efficiently computable pairings from the rest, we need to start with an introduction to the so-called **embedding degree** of a curve. add reference

Definition 5.2.0.1. **Embedding degree**
Let $\mathbb{F}$ be a finite field, let $E(\mathbb{F})$ be an elliptic curve over $\mathbb{F}$, and let n be a prime number that divides the order of $E(\mathbb{F})$. The embedding degree of $E(\mathbb{F})$ with respect to n is then the smallest integer k such that n divides $q^k - 1$.

Fermat's little theorem (page 21 ff.) implies that every curve has at least **some** embedding degree k, since at least $k = n-1$ is always a solution to the congruency $q^k \equiv 1 \pmod{n}$. This implies that the remainder of the integer division of $q^k - 1$ by n is 0. check reference

Example 81. To get a better intuition of the embedding degree, let's consider the elliptic curve $E_1(\mathbb{F}_5)$ from example 65. We know from 65 that the order of $E_1(\mathbb{F}_5)$ is 9, and, since the only prime factor of 9 is 3, we compute the embedding degree of $E_1(\mathbb{F}_5)$ with respect to 3. check reference

To find the embedding degree, we have to find the smallest integer k such that 3 divides $q^k - 1 = 5^k - 1$. We try and increment until we find a proper k. check reference

$$
\begin{aligned}
&k = 1\colon\ 5^1 - 1 = 4 && \text{not divisible by 3}\\
&k = 2\colon\ 5^2 - 1 = 24 && \text{divisible by 3}
\end{aligned}
$$

Now we know that the embedding degree of $E_1(\mathbb{F}_5)$ is 2 relative to the the prime factor 3.

Example 82. Let us consider the tiny-jubjub curve *TJJ_13* from example 66. We know from 66 that the order of *TJJ_13* is 20, and that the order therefore has two prime factors. A "large" prime factor 5 and a small prime factor 2. check reference

We start by computing the embedding degree of *TJJ_13* with respect to the large prime factor 5. To find that embedding degree, we have to find the smallest integer k such that 5 divides $q^k - 1 = 13^k - 1$. We try and increment until we find a proper k. check reference

$$
\begin{aligned}
&k = 1\colon\ 13^1 - 1 = 12 && \text{not divisible by 5}\\
&k = 2\colon\ 13^2 - 1 = 168 && \text{not divisible by 5}\\
&k = 3\colon\ 13^3 - 1 = 2196 && \text{not divisible by 5}\\
&k = 4\colon\ 13^4 - 1 = 28560 && \text{divisible by 5}
\end{aligned}
$$

Now we know that the embedding degree of *TJJ_13* is 4 relative to the the prime factor 5.

In real-world applications, like on pairing-friendly elliptic curves such as BLS_12-381, usually only the embedding degree of the large prime factor is relevant, which in the case of our tiny-jubjub curve is represented by 5. It should be noted, however that every prime factor of a curve's order has its own notation of embedding degree despite the fact that this is mostly irrelevant in applications.

To find the embedding degree of the small prime factor 2, we have to find the smallest integer k such that 2 divides $q^k - 1 = 13^k - 1$. We try and increment until we find a proper k.

$$k = 1\colon\ 13^1 - 1 = 12 \qquad\qquad \text{divisible by 2}$$

Now we know that the embedding degree of *TJJ_13* is 1 relative to the the prime factor 2. As we have seen, different prime factors can have different embedding degrees in general.

```
sage: p = 13
sage: # large prime factor
sage: n = 5
sage: for k in range(1,5): # Fermat's little theorem
....:     if (p^k-1)%n == 0:
....:         break
sage: k
4
sage: # small prime factor
sage: n = 2
sage: for k in range(1,2): # Fermat's little theorem
....:     if (p^k-1)%n == 0:
....:         break
sage: k
1
```

Example 83. To give an example of a cryptographically secure real-world elliptic curve that does not have a small embedding degree, let's look at curve Secp256k1 again. We know from 67 that the order of this curve is a prime number, so we only have a single embedding degree. check reference

To test potential embedding degrees k, say, in the range $1\ldots1000$, we can invoke Sage and compute as follows:

```
sage: p = 115792089237316195423570985008687907853269984665640
    564039457584007908834671663
sage: n = 115792089237316195423570985008687907852837564279074
    904382605163141518161494337
sage: for k in range(1,1000):
....:     if (p^k-1)%n == 0:
....:         break
sage: k
999
```

We see that Secp256k1 has at least no embedding degree $k < 1000$, which renders Secp256k1 a curve that has no small embedding degree. This property will be of importance later on.

Elliptic Curves over extension fields Suppose that p is a prime number, and $\mathbb{F}_p$ its associated prime field. We know from equation 4.29 that the fields $\mathbb{F}_{p^m}$ are extensions of $\mathbb{F}_p$ in the sense check reference that $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p^m}$. This implies that we can extend the affine plane that an elliptic curve is defined on by changing the base field to any extension field. To be more precise, let $E(\mathbb{F}) = \{(x,y) \in \mathbb{F}\times\mathbb{F} \mid y^2 = x^3 + a\cdot x + b\}$ be an affine short Weierstraß curve, with parameters a and b taken from $\mathbb{F}$. If $\mathbb{F}'$ is an extension field of $\mathbb{F}$, then we extend the domain of the curve by defining $E(\mathbb{F}')$ as follows:

$$E(\mathbb{F}') = \{(x,y) \in \mathbb{F}'\times\mathbb{F}' \mid y^2 = x^3 + a\cdot x + b\} \tag{5.23}$$

While we did not change the defining parameters, we consider curve points from the affine plane over the extension field now. Since $\mathbb{F} \subset \mathbb{F}'$, it can be shown that the original elliptic curve $E(\mathbb{F})$ is a sub-curve of the extension curve $E(\mathbb{F}')$.

Example 84. Consider the prime field $\mathbb{F}_5$ from example 59 and the elliptic curve $E_1(\mathbb{F}_5)$ from example 65. Since we know from XXX that $\mathbb{F}_{5^2}$ is an extension field of $\mathbb{F}_5$, we can extend the definition of $E_1(\mathbb{F}_5)$ to define a curve over $\mathbb{F}_{5^2}$:

check reference

check reference

$$E_1(\mathbb{F}_{5^2}) = \{(x,y) \in \mathbb{F}\times\mathbb{F} \mid y^2 = x^3 + x + 1\}$$

Since $\mathbb{F}_{5^2}$ contains 25 points, in order to compute the set $E_1(\mathbb{F}_{5^2})$, we have to try $25\cdot 25 = 625$ pairs, which is probably a bit too much for the average motivated reader. Instead, we invoke Sage to compute the curve for us. To do, we so choose the representation of $\mathbb{F}_{5^2}$ from XXX. We get:

add reference

add reference

```
sage: F5= GF(5)
sage: F5t.<t> = F5[]
sage: P = F5t(t^2+2)
sage: P.is_irreducible()
True
sage: F5_2.<t> = GF(5^2, name='t', modulus=P)
sage: E1F5_2 = EllipticCurve(F5_2,[1,1])
sage: E1F5_2.order()
27
```

The curve $E_1(\mathbb{F}_{5^2})$ consist of 27 points, in contrast to curve $E_1(\mathbb{F}_5)$, which consists of 9 points. Printing the points gives the following:

$$\begin{aligned} E_1(\mathbb{F}_{5^2}) = \{&\mathcal{O},(0,4),(0,1),(3,4),(3,1),(4,3),(4,2),(2,4),(2,1),\\ &(4t+3,3t+4),(4t+3,2t+1),(3t+2,t),(3t+2,4t),\\ &(2t+2,t),(2t+2,4t),(2t+1,4t+4),(2t+1,t+1),\\ &(2t+3,3),(2t+3,2),(t+3,2t+4),(t+3,3t+1),\\ &(3t+1,t+4),(3t+1,4t+1),(3t+3,3),(3t+3,2),(1,4t)\} \end{aligned}$$

As we can see, curve $E_1(\mathbb{F}_5)$ sits inside curve $E(\mathbb{F}_{5^2})$, which is implied by $\mathbb{F}_5$ being a subfield of $\mathbb{F}_{5^2}$.

Full torsion groups The fundamental theorem of finite cyclic groups XXX implies that every prime factor n of a cyclic group's order defines a subgroup of the size of the prime factor. Such a subgroup is called an n-torsion group. We have seen many of those subgroups in the examples XXX and XXX.

This needs to be written (in Algebra)

When we consider elliptic curve extensions as defined in 5.23, we could ask what happens to the n-torsion groups in the extension. One might intuitively think that their extension just

add reference

add reference

parallels the extension of the curve. For example, when $E(\mathbb{F}_p)$ is a curve over prime field $\mathbb{F}_p$, with some n-torsion group $\mathbb{G}$ and when we extend the curve to $E(\mathbb{F}_{p^m})$, then there is a bigger n-torsion group such that $\mathbb{G}$ is a subgroup. This might make intuitive sense, as $E(\mathbb{F}_p)$ is a sub-curve of $E(\mathbb{F}_{p^m})$.

However, the actual situation is a bit more surprising than that. To see that, let $\mathbb{F}_p$ be a prime field and let $E(\mathbb{F}_p)$ be an elliptic curve of order r, with embedding degree k and n-torsion group $E(\mathbb{F}_p)[n]$ for the same prime factor n of r. Then it can be shown that the n-torsion group $E(\mathbb{F}_{p^m})[n]$ of a curve extension is equal to $E(\mathbb{F}_p)[n]$, as long as the power m is less than the embedding degree k of $E(\mathbb{F}_p)$.

However, for the prime power p^m, for any $m \geq k$, $E(\mathbb{F}_{p^m})[n]$ is strictly larger than $E(\mathbb{F}_p)[n]$ and contains $E(\mathbb{F}_p)[n]$ as a subgroup. We call the n-torsion group $E(\mathbb{F}_{p^k})[n]$ of the extension of E over $\mathbb{F}_{p^k}$ the **full n-torsion group** of that elliptic curve. It can be shown that it contains n^2 many elements and consists of $n+1$ subgroups, one of which is $E(\mathbb{F}_p)[n]$.

So, roughly speaking, when we consider towers of curve extensions $E(\mathbb{F}_{p^m})$ ordered by the prime power m, then the n-torsion group stays constant for every level m, that is smaller than the embedding degree, while it suddenly blossoms into a larger group on level k with $n+1$ subgroups, and then stays like that for any level m larger than k. In other words, once the extension field is big enough to find one more point of order n (that is not defined over the base field), then we actually find all of the points in the full torsion group.

towers of curve extensions

Example 85. Consider curve $E_1(\mathbb{F}_5)$ again. We know that it contains a 3-torsion group and that the embedding degree of 3 is 2. From this we can deduce that we can find the full 3-torsion group $E_1[3]$ in the curve extension $E_1(\mathbb{F}_{5^2})$, the latter of which we computed in example 84 .

check reference

Since that curve is small, in order to find the full 3-torsion, we can loop through all elements of $E_1(\mathbb{F}_{5^2})$ and check check the defining equation $[3]P = \mathcal{O}$. Invoking Sage, we compute as follows:

```
sage: INF = E1F5_2(0) # Point at infinity
sage: L_E1_3 = []
sage: for p in E1F5_2:
....:     if 3*p == INF:
....:         L_E1_3.append(p)
sage: E1_3 = Set(L_E1_3) # Full 3-torsion set
```

We get the following result:

$$E_1[3] = \{\mathcal{O}, (1,t), (1,4t), (2,1), (2,4), (2t+1,t+1), (2t+1,4t+4), (3t+1,t+4), (3t+1,4t+1)\}$$

Example 86. Consider the tiny-jubjub curve from example 66. We know from example 82 that it contains a 5-torsion group and that the embedding degree of 5 is 4. This implies that we can find the full 5-torsion group *TJJ_13*[5] in the curve extension *TJJ_13*$(\mathbb{F}_{13^4})$.

check reference

check reference

To compute the full torsion, first observe that, since $\mathbb{F}_{13^4}$ contains 28561 elements, computing *TJJ_13*$(\mathbb{F}_{13^4})$ means checking $28561^2 = 815730721$ elements. From each of these curve points P, we then have to check the equation $[5]P = \mathcal{O}$. Doing this for 815730721 is a bit too slow even on a computer.

Fortunately, Sage has a way to loop through points of a given order efficiently. The following Sage code provides a way to compute the full torsion group:

```
sage: # define the extension field
sage: F13= GF(13) # prime field
sage: F13t.<t> = F13[] # polynomials over t
```

```
sage: P = F13t(t^4+2) # irreducible polynomial of degree 4
sage: P.is_irreducible()
True
sage: F13_4.<t> = GF(13^4, name='t', modulus=P) # F_{13^4}
sage: TJJF13_4 = EllipticCurve(F13_4,[8,8]) # tiny-jubjub
    extension
sage: # compute the full 5-torsion
sage: L_TJJF13_4_5 = []
sage: INF = TJJF13_4(0)
sage: for P in INF.division_points(5): # [5]P == INF
....:     L_TJJF13_4_5.append(P)
sage: len(L_TJJF13_4_5)
25
sage: TJJF13_4_5 = Set(L_TJJF13_4_5)
```

As expected, we get a group that contains $5^2 = 25$ elements. As it's rather tedious to write this group down, and as we don't need it in what follows, we forgo doing this. To see that the embedding degree 4 is actually the smallest prime power to find the full 5-torsion group, let's compute the 5-torsion group over of the tiny-jubjub curve of the extension field $\mathbb{F}_{13^3}$. We get the following:

```
sage: # define the extension field
sage: P = F13t(t^3+2) # irreducible polynomial of degree 3
sage: P.is_irreducible()
True
sage: F13_3.<t> = GF(13^3, name='t', modulus=P) # F_{13^3}
sage: TJJF13_3 = EllipticCurve(F13_3,[8,8]) # tiny-jubjub
    extension
sage: # compute the 5-torsion
sage: L_TJJF13_3_5 = []
sage: INF = TJJF13_3(0)
sage: for P in INF.division_points(5): # [5]P == INF
....:     L_TJJF13_3_5.append(P)
sage: len(L_TJJF13_3_5)
5
sage: TJJF13_3_5 = Set(L_TJJF13_3_5) # full $5$-torsion
```

As we can see, the 5-torsion group of tiny-jubjub over $\mathbb{F}_{13^3}$ is equal to the 5-torsion group of tiny-jubjub over $\mathbb{F}_{13}$ itself.

Example 87. Let's look at the curve Secp256k1. We know from example 67 that the curve is of some prime order r. Because of this, the only n-torsion group to consider is the curve itself, so the curve group is the r-torsion.

check reference

However, in order to find the full r-torsion of Secp256k1, we need to compute the embedding degree k. And as we have seen in XXX it is at least not small. However, we know from Fermat's little theorem (page 21 ff.) that a finite embedding degree must exist. It can be shown that it is given by the following 256-bit number:

add reference

$$k = 192986815395526992372618308347813175472927379845817397100860523586360249056$$

This means that the embedding degree is very large, which implies that the field extension $\mathbb{F}_{p^k}$ is very large too. To understand how big $\mathbb{F}_{p^k}$ is, recall that an element of $\mathbb{F}_{p^m}$ can be represented as a string $[x_0, \ldots, x_m]$ of m elements, each containing a number from the prime field $\mathbb{F}_p$. Now, in the case of Secp256k1, such a representation has k-many entries, each of them 256 bits in size. So, without any optimizations, representing such an element would need $k \cdot 256$ bits, which is too much to be represented in the observable universe.

Torsion subgroups As we have stated above, any full n-torsion group contains $n+1$ cyclic subgroups, two of which are of particular interest in pairing-based elliptic curve cryptography. To characterize these groups, we need to consider the so-called **Frobenius endomorphism** of an elliptic curve $E(\mathbb{F})$ over some finite field $\mathbb{F}$ of characteristic p:

$$\pi : E(\mathbb{F}) \to E(\mathbb{F}) : \begin{array}{ccc} (x,y) & \mapsto & (x^p, y^p) \\ \mathcal{O} & \mapsto & \mathcal{O} \end{array} \tag{5.24}$$

It can be shown that π maps curve points to curve points. The first thing to note is that, in case $\mathbb{F}$ is a prime field, the Frobenius endomorphism acts trivially, since $(x^p, y^p) = (x, y)$ on prime fields due to Fermat's little theorem (page 21 ff.). This means that the Frobenius map is more interesting over prime field extensions.

check reference

With the Frobenius map at hand, we can characterize two important subgroups of the full n-torsion. The first subgroup is the n-torsion group that already exists in the curve over the base field. In pairing-based cryptography, this group is usually written as $\mathbb{G}_1$, assuming that the prime factor n in the definition is implicitly given. Since we know that the Frobenius map acts trivially on curves over the prime field, we can define $\mathbb{G}_1$ as follows:

$$\mathbb{G}_1[n] := \{(x,y) \in E[n] \mid \pi(x,y) = (x,y)\ \} \tag{5.25}$$

In more mathematical terms, this definition means that $\mathbb{G}_1$ is the **Eigenspace** of the Frobenius map with respect to the **Eigenvalue** 1.

S: either add more explanation or move to a footnote

It can be shown that there is another subgroup of the full n-torsion group that can be characterized by the Frobenius map. In the context of so-called type 3 pairing-based cryptography, this subgroup is usually called $\mathbb{G}_2$ and it is defined as follows:

type 3 pairing-based cryptography

$$\mathbb{G}_2[n] := \{(x,y) \in E[n] \mid \pi(x,y) = [p](x,y)\ \} \tag{5.26}$$

In mathematical terms, $\mathbb{G}_2$ is the **Eigenspace** of the Frobenius map with respect to the **Eigenvalue** p.

Notation and Symbols 9. If the prime factor n of a curve's order is clear from the context, we sometimes simply write $\mathbb{G}_1$ and $\mathbb{G}_2$ to mean $\mathbb{G}_1[n]$ and $\mathbb{G}_2[n]$, respectively.

It should be noted, however that other definitions of $\mathbb{G}_2$ also exists in the literature. However, in the context of pairing-based cryptography, this is the most common one. It is particularly useful because we can define hash functions that map into $\mathbb{G}_2$, which is not possible for all subgroups of the full n-torsion.

add references?

Example 88. Consider the curve $E_1(\mathbb{F}_5)$ from example 65 again. As we have seen, this curve has the embedding degree $k = 2$, and a full 3-torsion group is given as follows:

$$\begin{aligned} E_1[3] = \{&\mathcal{O}, (2,1), (2,4), (1,t), (1,4t), (2t+1, t+1), \\ &(2t+1, 4t+4), (3t+1, t+4), (3t+1, 4t+1)\} \end{aligned} \tag{5.27}$$

According to the general theory, $E_1[3]$ contains 4 subgroups, and we can characterize the subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$ using the Frobenius endomorphism. Unfortunately, at the time of writing, Sage does not have a predefined Frobenius endomorphism for elliptic curves, so we have to use the Frobenius endomorphism of the underlying field as a temporary workaround. We compute as follows:

```
sage: L_G1 = []
sage: for P in E1_3:
....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
....:     if P == PiP:
....:         L_G1.append(P)
sage: G1 = Set(L_G1)
```

As expected, the group $\mathbb{G}_1 = \{\mathcal{O}, (2,4), (2,1)\}$ is identical to the 3-torsion group of the (unextended) curve over the prime field $E_1(\mathbb{F}_5)$. We can use almost the same algorithm to compute the group $\mathbb{G}_2$ and get the following:

```
sage: L_G2 = []
sage: for P in E1_3:
....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
....:     pP = 5*P # [5]P
....:     if pP == PiP:
....:         L_G2.append(P)
sage: G2 = Set(L_G2)
```

Thus, we have computed the the second subgroup of the full 3-torsion group of curve E_1 as the set $\mathbb{G}_2 = \{\mathcal{O}, (1,t), (1,4t)\}$.

Example 89. Consider the tiny-jubjub curve *TJJ_13* from example 66. In example 86, we computed its full 5 torsion, which is a group that has 6 subgroups. We compute $G1$ using Sage as follows:

check reference

check reference

```
sage: L_TJJ_G1 = []
sage: for P in TJJF13_4_5:
....:     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P)
....:     if P == PiP:
....:         L_TJJ_G1.append(P)
sage: TJJ_G1 = Set(L_TJJ_G1)
```

We get $\mathbb{G}1 = \{\mathcal{O}, (7,2), (8,8), (8,5), (7,11)\}$

```
sage: L_TJJ_G1 = []
sage: for P in TJJF13_4_5:
....:     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P)
....:     pP = 13*P # [5]P
....:     if pP == PiP:
....:         L_TJJ_G1.append(P)
sage: TJJ_G1 = Set(L_TJJ_G1)
```

$\mathbb{G}_2 = \{\mathcal{O}, (9t^2+7, t^3+11t,), (9t^2+7, 12t^3+2t), (4t^2+7, 5t^3+10t), (4t^2+7, 8t^3+3t)\}$

Example 90. Consider Bitcoin's curve Secp256k1 again. Since the group $\mathbb{G}_1$ is identical to the torsion group of the unextended curve, and since Secp256k1 has prime order, we know that, in this case, $\mathbb{G}_1$ is identical to Secp256k1. It is however, infeasible not to compute not only $\mathbb{G}_2$ itself, but to even compute an average element of $\mathbb{G}_2$, as elements need too much storage to be representable in this universe.

The Weil pairing In this part, we consider a pairing function defined on the subgroups $\mathbb{G}_1[r]$ and $\mathbb{G}_2[r]$ of the full r-torsion $E[r]$ of a short Weierstraß elliptic curve. To be more precise, let $E(\mathbb{F}_p)$ be an elliptic curve of embedding degree k such that r is a prime factor of its order. Then the **Weil pairing** is a bilinear, non-degenerate map:

$$e(\cdot,\cdot) : \mathbb{G}_1[r] \times \mathbb{G}_2[r] \to \mathbb{F}_{p^k} \; ; \; (P,Q) \mapsto (-1)^r \cdot \frac{f_{r,P}(Q)}{f_{r,Q}(P)} \tag{5.28}$$

The extension field elements $f_{r,P}(Q), f_{r,Q}(P) \in \mathbb{F}_{p^k}$ are computed by **Miller's algorithm**: check floating of algorithm

Algorithm 7 Miller's algorithm for short Weierstraß curves $y^2 = x^3 + ax + b$

Require: $r > 3$, $P \in E[r]$, $Q \in E[r]$ and
  $b_0, \ldots, b_t \in \{0,1\}$ with $r = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_t \cdot 2^t$ and $b_t = 1$
  procedure MILLER'S ALGORITHM(P,Q)
    if $P = \mathcal{O}$ or $Q = \mathcal{O}$ or $P = Q$ **then**
      return $f_{r,P}(Q) \leftarrow (-1)^r$
    end if
    $(x_T, y_T) \leftarrow (x_P, y_P)$
    $f_1 \leftarrow 1$
    $f_2 \leftarrow 1$
    for $j \leftarrow t-1, \ldots, 0$ **do**
      $m \leftarrow \frac{3 \cdot x_T^2 + a}{2 \cdot y_T}$
      $f_1 \leftarrow f_1^2 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$
      $f_2 \leftarrow f_2^2 \cdot (x_Q + 2x_T - m^2)$
      $x_{2T} \leftarrow m^2 - 2x_T$
      $y_{2T} \leftarrow -y_T - m \cdot (x_{2T} - x_T)$
      $(x_T, y_T) \leftarrow (x_{2T}, y_{2T})$
      if $b_j = 1$ **then**
        $m \leftarrow \frac{y_T - y_P}{x_T - x_P}$
        $f_1 \leftarrow f_1 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$
        $f_2 \leftarrow f_2 \cdot (x_Q + (x_P + x_T) - m^2)$
        $x_{T+P} \leftarrow m^2 - x_T - x_P$
        $y_{T+P} \leftarrow -y_T - m \cdot (x_{T+P} - x_T)$
        $(x_T, y_T) \leftarrow (x_{T+P}, y_{T+P})$
      end if
    end for
    $f_1 \leftarrow f_1 \cdot (x_Q - x_T)$
    return $f_{r,P}(Q) \leftarrow \frac{f_1}{f_2}$
  end procedure

Understanding how the algorithm works in detail requires the concept of **divisors**, which is outside of the scope this book. The interested reader might look at XXX. add references

In real-world applications of pairing-friendly elliptic curves, the embedding degree is usually a small number like 2, 4, 6 or 12, and the number r is the largest prime factor of the curve's order.

Example 91. Consider curve $E_1(\mathbb{F}_5)$ from example 65. Since the only prime factor of the group's order is 3, we cannot compute the Weil pairing on this group using our definition of Miller's algorithm. In fact, since $\mathbb{G}_1$ is of order 3, executing the if statement on line XXX will lead to a "division by zero" error in the computation of the slope m.

check reference

add reference

Example 92. Consider the tiny-jubjub curve *TJJ_13*$(\mathbb{F}_{13})$ from example 66 again. We want to instantiate the general definition of the Weil pairing for this example. To do so, recall that, as we have see in example 82, its embedding degree is 4, and that we have the following type-3 pairing groups (where $\mathbb{G}_1$ and $\mathbb{G}_2$ are subgroups of the full 5-torsion found in the curve *TJJ_13*$(\mathbb{F}_{13^4})$):

check reference

check reference

$$\mathbb{G}_1 = \{\mathcal{O}, (7,2), (8,8), (8,5), (7,11)\}$$
$$\mathbb{G}_2 = \{\mathcal{O}, (9t^2+7, t^3+11t), (9t^2+7, 12t^3+2t), (4t^2+7, 5t^3+10t), (4t^2+7, 8t^33+3t)\}$$

The type-3 Weil pairing is a map $e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_{13^4}$. From the first if-statement in Miller's algorithm, we can deduce that $e(\mathcal{O}, Q) = 1$ as well as $e(P, \mathcal{O}) = 1$ for all arguments $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$. In order to compute a non-trivial Weil pairing, we choose the arguments $P = (7,2)$ and $Q = (9t^2+7, 12t^3+2t)$.

To compute the pairing $e((7,2), (9t^2+7, 12t^3+2t))$, we have to compute the extension field elements $f_{5,P}(Q)$ and $f_{5,Q}(P)$ by applying Miller's algorithm. Do do so, observe that we have $5 = 1\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2$, so we get $t = 2$ as well as $b_0 = 1$, $b_1 = 0$ and $b_2 = 1$. The loop therefore needs to be executed two times.

Computing $f_{5,P}(Q)$, we initiate $(x_T, y_T) = (7,2)$ as well as $f_1 = 1$ and $f_2 = 1$. Then we proceed as follows:

j	b_j	m	f_1	f_2	x_{2T}	y_{2T}	x_{T+P}	y_{T+P}
1	$\cdot$							

$$
\begin{aligned}
m &= \frac{3 \cdot x_T^2 + a}{2 \cdot y_T} \\
&= \frac{3 \cdot 2^2 + 1}{2 \cdot 4} = \frac{3}{3} \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
f_1 &= f_1^2 \cdot (y_Q - y_T - m \cdot (x_Q - x_T)) \\
&= 1^2 \cdot (t - 4 - 1 \cdot (1 - 2)) = t - 4 + 1 \\
&= t + 2
\end{aligned}
$$

$$
\begin{aligned}
f_2 &= f_2^2 \cdot (x_Q + 2x_T - m^2) \\
&= 1^2 \cdot (1 + 2 \cdot 2 - 1^2) = (1 + 4 - 1) \\
&= 4
\end{aligned}
$$

$$
\begin{aligned}
x_{2T} &= m^2 - 2x_T \\
&= 1^2 - 2 \cdot 2 = -3 \\
&= 2
\end{aligned}
$$

$$
\begin{aligned}
y_{2T} &= -y_T - m \cdot (x_{2T} - x_T) \\
&= -4 - 1 \cdot (2 - 2) = -4 \\
&= 1
\end{aligned}
$$

We update $(x_T, y_T) = (2, 1)$ and, since $b_0 = 1$, we have to execute the if statement on line XXX in the **for** loop. However, since we only loop a single time, we don't need to compute y_{T+P}, since we only need the updated x_T in the final step. We get:

add reference

should all lines of all algorithms be numbered?

$$
\begin{aligned}
m &= \frac{y_T - y_P}{x_T - x_P} \\
&= \frac{1 - 4}{2 - x_P}
\end{aligned}
$$

$$
f_1 = f_1 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))
$$

$$
f_2 = f_2 \cdot (x_Q + (x_P + x_T) - m^2)
$$

$$
x_{T+P} = m^2 - x_T - x_P
$$

5.3 Hashing to Curves

Elliptic curve cryptography frequently requires the ability to hash data onto elliptic curves. If the order of the curve is not a prime number, hashing to prime number subgroups is also of importance. In the context of pairing-friendly curves, it is also sometimes necessary to hash specifically onto the group $\mathbb{G}_1$ or $\mathbb{G}_2$.

As we have seen in section 4.1.2, many general methods are known for hashing into groups in general, and finite cyclic groups in particular. As elliptic groups are cyclic, those methods can be utilized in this case, too. However, in what follows we want to describe some methods specific to elliptic curves that are frequently used in real-world applications.

check reference

Try-and-increment hash functions One of the most straight-forward ways of hashing a bit-string onto an elliptic curve point in a secure way is to use a cryptographic hash function together with one of the methods we described in section 4.1.2 to hash to the modular arithmetics base field of the curve. Ideally, the hash function generates an image that is at least one bit longer than the bit representation of the base field modulus.

check reference

The image in the base field can then be interpreted as the x coordinate of the curve point, and the two possible y coordinates are derived from the curve equation, while one of the bits that exceeded the modulus determines which of the two y coordinates to choose.

Such an approach would be deterministic and easy to implement, and it would conserve the cryptographic properties of the original hash function. However, not all x coordinates generated in such a way will result in quadratic residues when inserted into the defining equation. It follows that not all field elements give rise to actual curve points. In fact, on a prime field, only half of the field elements are quadratic residues. Hence, assuming an even distribution of the hash values in the field, this method would fail to generate a curve point in about half of the attempts.

One way to account for this problem is the so-called **try-and-increment** method. Its basic assumption is that, when hashing different values, the result will eventually lead to a valid curve point.

Therefore, instead of simply hashing a string s to the field, we hash the concatenation of s with additional bytes to the field instead. In other words, we use a try-and-increment hash as described in 5. If the first try of hashing to the field does not result in a valid curve point, the counter is incremented, and the hashing is repeated again. This is done until a valid curve point is found.

check reference

check if the algorithm is floated properly

This method has a number of advantages: It is relatively easy to implement in code, and it maintains the cryptographic properties of the original hash function. However, it is not guaranteed to find a valid curve point, as there is a chance that all possible values in the chosen size of the counter will fail to generate a quadratic residue. Fortunately, it is possible to make the probability for this arbitrarily small by choosing large enough counters and relying on the (approximate) uniformity of the hash-to-field function.

If the curve is not of prime order, the result will be a general curve point that might not be in the "large" prime-order subgroup. In this case, a **cofactor clearing** step is then necessary to project the curve point onto the subgroup. This is done by scalar multiplication with the cofactor of prime order with respect to the curves order.

Example 93. Consider the tiny-jubjub curve from example 66. We want to construct a try-and-increment hash function that hashes a binary string s of arbitrary length onto the large prime-order subgroup of size 5.

check reference

Since the curve, as well as our targeted subgroup, is defined over the field $\mathbb{F}_{13}$, and the binary representation of 13 is $13.bits() = 1101$, we apply SHA256 from Sage's hashlib library on the concatenation $s||c$ for some binary counter string, and use the first 4 bits of the image to try to hash into $\mathbb{F}_{13}$. In case we are able to hash to a value z such that $z^3 + 8 \cdot z + 8$ is a quadratic residue in $\mathbb{F}_{13}$, we use the 5-th bit to decide which of the two possible roots of $z^3 + 8 \cdot z + 8$ we

Algorithm 8 Hash-to-$E(\mathbb{F}_r)$

Require: $r \in \mathbb{Z}$ with $r.nbits() = k$ and $s \in \{0,1\}^*$
Require: Curve equation $y^2 = x^3 + ax + b$ over $\mathbb{F}_r$
  procedure TRY-AND-INCREMENT(r,k,s)
    $c \leftarrow 0$
    repeat
      $s' \leftarrow s||c\_bits()$
      $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \ldots + H(s')_k \cdot 2^k$
      $x \leftarrow z^3 + a \cdot z + b$
      $c \leftarrow c + 1$
    until $z < r$ and $x^{\frac{r-1}{2}} \bmod r = 1$
    if $H(s')_{k+1} == 0$ **then**
      $y \leftarrow \sqrt{x}$ #(root in $\mathbb{F}_r$)
    else
      $y \leftarrow r - \sqrt{x}$ #(root in $\mathbb{F}_r$)
    end if
    return (x,y)
  end procedure
Ensure: $(x,y) \in E(\mathbb{F}_r)$

will choose as the y coordinate. The result is a curve point different from the point at infinity. To project it to a point of $\mathbb{G}_1$, we multiply it with the cofactor 4. If the result is still not the point at infinity, it is the result of the hash.

To make this concrete, let $s =' 10011001111010110100000111'$ be our binary string that we want to hash onto $\mathbb{G}_1$. We use a 4-bit binary counter starting at zero, that is, we choose $c = 0000$. Invoking Sage, we define the try-hash function as follows:

```
sage: import hashlib
sage: def try_hash(s,c):
....:     s_1 = s+c
....:     hasher = hashlib.sha256(s_1.encode('utf-8'))
....:     digest = hasher.hexdigest()
....:     d = Integer(digest,base=16)
....:     sign = d.str(2)[-5:-4]
....:     d = d.str(2)[-4:]
....:     z = Integer(d,base=2)
....:     return (z,sign)
sage: try_hash('10011001111010110100000111','0000')
(15, '1')
```

As we can see, our first attempt to hash into $\mathbb{F}_{13}$ was not successful, as 15 is not a number in $\mathbb{F}_{13}$, so we increment the binary counter by 1 and try again:

```
sage: try_hash('10011001111010110100000111','0001')
(3, '0')
```

With this try, we found a hash into $\mathbb{F}_{13}$. However, this point is not guaranteed to define a curve point. To see that, we insert $z = 3$ into the right side of the Weierstraß equation of the tiny-jubjub curve, and compute $3^3 + 8 * 3 + 8 = 7$. However, 7 is not a quadratic residue in

$\mathbb{F}_{13}$, since $7^{\frac{13-1}{2}} = 7^6 = 12 = -1$. This means that 3 is a not a suitable point, and we have to increment the counter two more times:

```
sage: try_hash('100110011110101101000000111','0010')
(3, '0')
sage: try_hash('100110011110101101000000111','0011')
(6, '1')
```

Since $6^3+8\cdot 6+8 = 12$, and we have $\sqrt{12} \in \{5,8\}$, we finally found the valid x coordinate $x = 6$ for the curve point hash. Now, since the sign bit of this hash is 1, we choose the larger root $y = 8$ as the y coordinate and get the following hash which is a valid curve point point on the tiny-jubjub curve:

$$H('100110011110101101000000111') = (6,8)$$

In order to project this onto the "large" prime-order subgroup, we have to do cofactor clearing, that is, we have to multiply the point with the cofactor 4. We get the following:

$$[4](6,8) = \mathcal{O}$$

This means that the hash value is still not right. We therefore have to increment the counter two more times again, until we finally find a correct hash to $\mathbb{G}_1$:

```
sage: try_hash('100110011110101101000000111','0100')
(0, '1')
sage: try_hash('100110011110101101000000111','0101')
(12, '0')
```

Since $12^3+8\cdot 12+8 = 12$, and we have $\sqrt{12} \in \{5,8\}$, we found another valid x coordinate $x = 12$ for the curve point hash. Since the sign bit of this hash is 0, we choose the smaller root $y = 5$ as the y coordinate, and get the following hash, which is a valid curve point point on the tiny-jubjub curve:

$$H('100110011110101101000000111') = (12,5)$$

In order to project this onto the "large" prime-order subgroup we have to do cofactor clearing, that is, we have to multiply the point with the cofactor 4. We get the following: again?

$$[4](12,5) = (8,5)$$

So, hashing the binary string $'100110011110101101000000111'$ onto $\mathbb{G}_1$ gives the hash value $(8,5)$ as a result.

5.4 Constructing elliptic curves

Cryptographically secure elliptic curves like Secp256k1 from example 67 have been known for quite some time. Given the latest advancements of cryptography, however, it is often necessary to design and instantiate elliptic curves from scratch that satisfy certain very specific properties. check reference

For example, in the context of SNARK development, it was necessary to design a curve that can be efficiently implemented inside of a so-called circuit in order to enable primitives like elliptic curve signature schemes in a zero-knowledge proof. Such a curve is given by the Baby-jubjub curve in XXX, and we have paralleled its definition by introducing the tiny-jubjub curve circuit signature schemes add reference from example 66. As we have seen, those curves are instances of so-called twisted Edwards curves, and as such have easy to implement addition laws that work without branching. However, we introduced the tiny-jubjub curve out of thin air, as we just gave the curve parameters without explaining how we came up with them.

check reference

Another requirement in the context of many so-called **pairing-based zero-knowledge proofing systems** is the existence of a suitable, pairing-friendly curve with a specified security level and a low embedding degree as defined in 5.2.0.1. Famous examples are the BLS_12 and the NMT curves.

check reference

The major goal of this section is to explain the most important method of designing elliptic curves with predefined properties from scratch, called the **complex multiplication method**. We will apply this method in section XXX to synthesize a particular BLS_6 curve, which is one of the most insecure curves, but it will serve as the main curve to build our pen-and-paper SNARKs on. As we will see, this curve has a "large" prime factor subgroup of order 13, which implies that we can use our tiny-jubjub curve to implement certain elliptic curve cryptographic primitives in circuits over that BLS_6 curve.

add references

add reference

Before we introduce the complex multiplication method, we have to explain a few properties of elliptic curves that are of key importance in understanding the complex multiplication method.

The Trace of Frobenius To understand the complex multiplication method of elliptic curves, we have to define the so-called **trace** of an elliptic curve first.

We know from XXX that elliptic curves over finite fields are products of cyclic groups of finite order. Therefore, an interesting question is whether it is possible to estimate the number of elements that this curve contains. Since an affine short Weierstraß curve consists of pairs (x,y) of elements from a finite field $\mathbb{F}_q$ plus the point at infinity, and the field $\mathbb{F}_q$ contains q elements, the number of curve points cannot be arbitrarily large, since it can contain at most q^2+1 many elements.

reference text to be written in Algebra

There is however, a more precise estimation, usually called the **Hasse bound**. To understand it, let $E(\mathbb{F}_q)$ be an affine short Weierstraß curve over a finite field $\mathbb{F}_w$ of order q, and let $|E(\mathbb{F}_q)|$ be the order of the curve. Then there is an integer $t \in \mathbb{Z}$, called the **trace of Frobenius** of the curve, such that $|t| \leq 2\sqrt{q}$ and the following equation holds:

$$|E(\mathbb{F})| = q+1-t \tag{5.29}$$

A positive trace, therefore, implies that the curve contains less points than the underlying field, whereas a negative trace means that the curve contains more points. However, the estimation $|t| \leq 2\sqrt{q}$ implies that the difference is not very large in either direction, and the number of elements in an elliptic curve is always approximately in the same order of magnitude as the size of the curve's base field.

Example 94. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example 65. We know that it contains 9 curve points. Since the order of $\mathbb{F}_5$ is 5, we compute the trace of $E_1(\mathbb{F})$ to be $t=-3$, since the Hasse bound is given by the following equation:

check reference

$$9 = 5+1-(-3)$$

Indeed, we have $|t| \leq 2\sqrt{q}$, since $\sqrt{5} > 2.23$ and $|-3| = 3 \leq 4.46 = 2\cdot 2.23 < 2\cdot\sqrt{5}$.

Example 95. To compute the trace of the tiny-jubjub curve, recall from example 74 that the order of *TJJ_13* is 20. Since the order of $\mathbb{F}_{13}$ is 13, we can therefore use the Hasse bound and compute the trace as $t=-6$:

check reference

$$20 = 13+1-(-6) \tag{5.30}$$

Again, we have $|t| \leq 2\sqrt{q}$, since $\sqrt{13} > 3.60$ and $|-6| = 6 \leq 7.20 = 2 \cdot 3.60 < 2 \cdot \sqrt{13}$.

Example 96. To compute the trace of Secp256k1, recall from example 67 that this curve is defined over a prime field with p elements, and that the order of that group is given by r:

check reference

$$
\begin{aligned}
p &= 115792089237316195423570985008687907853269984665640564039457584007908834671663 \\
r &= 115792089237316195423570985008687907852837564279074904382605163141518161494337
\end{aligned}
$$

Using the Hesse bound $r = p + 1 - t$, we therefore compute $t = p + 1 - r$, which gives the trace of curve Secp256k1 as follows:

$$t = 432420386565659656852420866390673177327$$

As we can see, Secp256k1 contains less elements than its underlying field. However, the difference is tiny, since the order of Secp256k1 is in the same order of magnitude as the order of the underlying field. Compared to p and r, t is tiny.

```
sage: p = 1157920892373161954235709850086879078532699846656405
    64039457584007908834671663
sage: r = 115792089237316195423570985008687907852837564279074
    904382605163141518161494337
sage: t = p + 1 -r
sage: t.nbits()
129
sage: abs(RR(t)) <= 2*sqrt(RR(p))
True
```

The j-invariant As we have seen in XXX, two elliptic curves $E_1(\mathbb{F})$ defined by $y^2 = x^3 + ax + b$ and $E_2(\mathbb{F})$ defined by $y^2 + a'x + b'$ are strictly isomorphic if and only if there is a quadratic residue $d \in \mathbb{F}$ such that $a' = ad^2$ and $b' = bd^3$.

add reference

There is, however, a more general way to classify elliptic curves over finite fields $\mathbb{F}_q$, based on the so-called **j-invariant** of an elliptic curve with $j(E(\mathbb{F}_q)) \in \mathbb{F}_q$, as defined below:

$$j(E(\mathbb{F}_q)) = (1728 \bmod q) \frac{4 \cdot a^3}{4 \cdot a^3 + (27 \bmod q) \cdot b^2} \tag{5.31}$$

A detailed description of the j-invariant is beyond the scope of this book. For our present purposes, it is sufficient to note that two elliptic curves $E_1(\mathbb{F})$ and $E_2(\mathbb{F}')$ are isomorphic over the algebraic closures of $\mathbb{F}$ and $\mathbb{F}'$, if and only if $\overline{\mathbb{F}} = \overline{\mathbb{F}'}$ and $j(E_1) = j(E_2)$.

algebraic closures

So, the j-invariant is an important tool to classify elliptic curves and it is needed in the complex multiplication method to decide on an actual curve instantiation that implements abstractly chosen properties.

Example 97. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example 65. We compute its j-invariant as follows:

check reference

$$
\begin{aligned}
j(E_1(\mathbb{F}_5)) &= (1728 \bmod 5) \frac{4 \cdot 1^3}{4 \cdot 1^3 + (27 \bmod 5) \cdot 1^2} \\
&= 3 \frac{4}{4+2} \\
&= 3 \cdot 4 & = 2
\end{aligned}
$$

Example 98. Consider the elliptic curve *TJJ_13* from example 66. We compute its j-invariant as follows: check reference

$$
\begin{aligned}
j(E_1(\mathbb{F}_5)) &= (1728 \bmod 13)\,\frac{4\cdot 8^3}{4\cdot 8^3+(27 \bmod 13)\cdot 8^2}\\
&= 12\cdot\frac{4\cdot 5}{4\cdot 5+1\cdot 12}\\
&= 12\cdot\frac{7}{7+12}\\
&= 12\cdot 7\cdot 6^{-1}\\
&= 12\cdot 7\cdot 11\\
01
\end{aligned}
$$

Example 99. Consider Secp256k1 from example Secp256k1. We compute its j-invariant using Sage: check reference

```
sage: p = 1157920892373161954235709850086879078532699846656405
    64039457584007908834671663
sage: F = GF(p)
sage: j = F(1728)*((F(4)*F(0)^3)/(F(4)*F(0)^3+F(27)*F(7)^2))
sage: j == F(0)
True
```

The Complex Multiplication Method As we have seen in the previous sections, elliptic curves have various defining properties, like their order, their prime factors, the embedding degree, or the cardinality (number of elements) of the base field. The **complex multiplication** (CM) method provides a practical way of constructing elliptic curves with pre-defined restrictions on the order and the base field.

The method usually starts by choosing a base field $\mathbb{F}_q$ of the curve $E(\mathbb{F}_q)$ we want to construct such that $q = p^m$ for some prime number p, and “ $m \in \mathbb{N}$ with $m \geq 1$. We assume $p > 3$ to simplify things in what follows.

Next, the trace of Frobenius $t \in \mathbb{Z}$ of the curve is chosen such that p and t are coprime, that is, $gcd(p,t) = 0$ holds true. The choice of t also defines the curve's order r, since $r = p+1-t$ by the Hasse bound (equation 5.29), so choosing t will define the large order subgroup as well as all small cofactors. r has to be defined in such a way that the elliptic curve meets the security requirements of the application it is designed for. check reference

Note that the choice of p and t also determines the embedding degree k of any prime-order subgroup of the curve, since k is defined as the smallest number such that the prime order n divides the number $q^k - 1$.

$$
\begin{aligned}
D &< 0\\
D \bmod 4 = 0 \text{ or } D &\bmod 4 = 1\\
4q &= t^2 + |D|v^2
\end{aligned}
\tag{5.32}
$$

In order for the complex multiplication method to work, neither q nor t can be arbitrary, but must be chosen in such a way that two additional integers $D \in \mathbb{Z}$ and $v \in \mathbb{Z}$ exist and the following conditions hold:

If such numbers exist, we call D the **CM-discriminant**, and we know that we can construct a curve $E(\mathbb{F}_q)$ over a finite field $\mathbb{F}_q$ such that the order of the curve is $|E(\mathbb{F}_q)| = q+1-t$.

It is the content of the complex multiplication method to actually construct such a curve, that is finding the parameters a and b from $\mathbb{F}_q$ in the defining Weiertraß equation such that the curve has the desired order r.

Finding solutions to equation 5.29,= can be achieved in different ways, but we will forego the fine detail here. In general, it can be said that there are well-known constraints for elliptic curve families (e.g. the BLS (ECT) families) that provides families of solutions. In what follows, we will look at one type curve in the BLS-family, which gives an entire range of solutions.Are we looking at a subtype of BLS or is BLS the specific type we're referring to?

check reference

disambiguate

Assuming that the proper parameters q, t, D and v are found, we have to compute the so-called **Hilbert class polynomial** $H_D \in \mathbb{Z}[x]$ of the CM-discriminant D, which is a polynomial with integer coefficients. To do so, we first have to compute the following set:

$$
\begin{aligned}
ICG(D) = \{(A,B,C) \mid A,B,C \in \mathbb{Z}, D = B^2-4AC, gcd(A,B,C) = 1,\\
|B| \leq A \leq \sqrt{\frac{|D|}{3}}, A \leq C, \text{ if } B < 0 \text{ then } |B| < A < C\}
\end{aligned}
$$

One way to compute this set is to first compute the integer $A_{max} = Floor(\sqrt{\frac{|D|}{3}})$, then loop through all the integers A in the range $[0,\ldots,A_{max}]$, as well as through all the integers B in the range $[-A_{max},\ldots,A_{max}]$, then see if there is an integer C that satisfies $D = B^2-4AC$ and the rest of the requirements in XXX.

add reference

To compute the Hilbert class polynomial, the so-called ***j*-function** (or j-invariant) is needed, which is a complex function defined on the upper half $\mathbb{H}$ of the complex plane $\mathbb{C}$, usually written as follows:is this the same as equation eq:j-invariant1 No they just sound very similar. But they are very different?

unify terminology

$$
j : \mathbb{H} \to \mathbb{C} \tag{5.33}
$$

Roughly speaking, what this means is that the j-functions takes complex numbers $(x+i\cdot y)$ with a positive imaginary part $y > 0$ as inputs and returns a complex number $j(x+i\cdot y)$ as a result.

For the purposes of this book, it is not important to understand the j-function in detail, and we can use Sage to compute it in a similar way that we would use Sage to compute any other well-known function. It should be noted, however, that the computation of the j-function in Sage is sometimes prone to precision errors. For example, the j-function has a root in $\frac{-1+i\sqrt{3}}{2}$, which Sage only approximates. Therefore, when using Sage to compute the j-function, we need to take precision loss into account and possibly round to the nearest integer.

```
sage: z = ComplexField(100)(0,1)
sage: z # (0+1i)
1.0000000000000000000000000000*I
sage: elliptic_j(z)
1728.0000000000000000000000000
sage: # j-function only defined for positive imaginary
   arguments
sage: z = ComplexField(100)(1,-1)
sage: try:
....:     elliptic_j(z)
....: except PariError:
```

```
....:     pass
sage: # root at (-1+i sqrt(3))/2
sage: z = ComplexField(100)(-1,sqrt(3))/2
sage: elliptic_j(z)
-2.6445453750358706361219364880e-88
sage: elliptic_j(z).imag().round()
0
sage: elliptic_j(z).real().round()
0
```

With a way to compute the j-function and the precomputed set $ICG(D)$ at hand, we can now compute the Hilbert class polynomial as follows:

$$H_D(x) = \Pi_{(A,B,C)\in ICG(D)}\left(x - j\left(\frac{-B+\sqrt{D}}{2A}\right)\right) \tag{5.34}$$

In other words, we loop over all elements (A,B,C) from the set $ICG(D)$ and compute the j-function at the point $\frac{-B+\sqrt{D}}{2A}$, where D is the CM-discriminant that we chose in a previous step. The result defines a factor of the Hilbert class polynomial and all factors are multiplied together.

It can be shown that the Hilbert class polynomial is an integer polynomial, but actual computations need high-precision arithmetics to avoid approximation errors that usually occur in computer approximations of the j-function (as shown above). So, in case the calculated Hilbert class polynomial does not have integer coefficients, we need to round the result to the nearest integer. Given that the precision we used was high enough, the result will be correct.

In the next step, we use the Hilbert class polynomial $H_D \in \mathbb{Z}[x]$, and project it to a polynomial $H_{D,q} \in \mathbb{F}_q[x]$ with coefficients in the base field $\mathbb{F}_q$ as chosen in the first step. We do this by simply computing the new coefficients as the old coefficients modulus p, that is, if $H_D(x) = a_m x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$, we compute the q-modulus of each coefficient $\tilde{a}_j = a_j \bmod p$, which defines the **projected Hilbert class polynomial** as follows:

$$H_{D,p}(x) = \tilde{a}_m x^m + \tilde{a}_{m-1}x^{m-1} + \ldots + \tilde{a}_1 x + \tilde{a}_0$$

We then search for roots of $H_{D,p}$, since every root j_0 of $H_{D,p}$ defines a family of elliptic curves over $\mathbb{F}_q$, which all have a j-invariant 5.31 or 5.33 equal to j_0. We can pick any root, since all of them will lead to proper curves eventually. check reference

However, some of the curves with the correct j-invariant might have an order different from the one we initially decided on. Therefore, we need a way to decide on a curve with the correct order.

To compute such a curve, we have to distinguish a few different cases based on our choice of the root j_0 and of the CM-discriminant D. If $j_0 \neq 0$ or $j_0 \neq 1728 \bmod q$, we compute $c_1 = \frac{j_0}{(1728 \bmod q) - j_0}$, then we chose some arbitrary quadratic non-residue $c_2 \in \mathbb{F}_q$, and some arbitrary cubic non-residue $c_3 \in \mathbb{F}_q$.

The following table is guaranteed to define a curve with the correct order $r = q+1-t$ for the trace of Frobenius t we initially decided on: actually make this a table?

Definition 5.4.0.1. • Case $j_0 \neq 0$ and $j_0 \neq 1728 \bmod q$. A curve with the correct order is defined by one of the following equations:

$$y^2 = x^3 + 3c_1 x + 2c_1 \quad \text{or} \quad y^2 = x^3 + 3c_1c_2^2 x + 2c_1c_2^3 \tag{5.35}$$

- Case $j_0 = 0$ and $D \neq -3$. A curve with the correct order is defined by one of the following equations:
$$y^2 = x^3 + 1 \quad \text{or} \quad y^2 = x^3 + c_2^3 \tag{5.36}$$

- Case $j_0 = 0$ and $D = -3$. A curve with the correct order is defined by one of the following equations:
$$\begin{array}{ccc} y^2 = x^3 + 1 & \text{or} & y^2 = x^3 + c_2^3 \quad \text{or} \\ y^2 = x^3 + c_3^2 & \text{or} & y^2 = c_3^2 c_2^3 \quad \text{or} \\ y^2 = x^3 + c_3^{-2} & \text{or} & y^2 = x^3 + c_3^{-2} c_2^3 \end{array}$$

- Case $j_0 = 1728 \bmod q$ and $D \neq -4$. A curve with the correct order is defined by one of the following equations:
$$y^2 = x^3 + x \quad \text{or} \quad y^2 = x^3 + c_2^2 x \tag{5.37}$$

- Case $j_0 = 1728 \bmod q$ and $D = -4$. A curve with the correct order is defined by one of the following equations:
$$\begin{array}{ccc} y^2 = x^3 + x & \text{or} & y^2 = x^3 + c_2 x \quad \text{or} \\ y^2 = x^3 + c_2^2 x & \text{or} & y^2 = x^3 + c_2^3 x \end{array}$$

To decide the proper defining Weierstraß equation, we therefore have to compute the order of any of the potential curves above, and then choose the one that fits our initial requirements. Since it can be shown that the Hilbert class polynomials for the CM-discriminants $D = -3$ and $D = -4$ are given by $H_{-3,q}(x) = x$ and $H_{-4,q} = x - (1728 \bmod q)$ (EXERCISE), the previous cases are exhaustive.

exercise still to be written?

To summarize, using the complex multiplication method, it is possible to synthesize elliptic curves with predefined order over predefined base fields from scratch. However, the curves that are constructed this way are just some representatives of a larger class of curves, all of which have the same order. Therefore, in real-world applications, it is sometimes more advantageous to choose a different representative from that class. To do so recall from XXX that any curve defined by the Weierstraß equation $y^2 = x^3 + axb$ is isomorphic to a curve of the form $y^2 = x^3 + ad^2x + bd^3$ for some quadratic residue $d \in \mathbb{F}_q$.

add reference

In order to find a suitable representative (e.g. with small parameters a and b) in the last step, the curve designer might choose a quadratic residue d such that the transformed curve has the properties they wanted.

Example 100. Consider curve $E_1(\mathbb{F}_5)$ from example 65. We want to use the complex multiplication method to derive that curve from scratch. Since $E_1(\mathbb{F}_5)$ is a curve of order $r = 9$ over the prime field of order $q = 5$, we know from example 94 that its trace of Frobenius is $t = -3$, which also implies that q and $|t|$ are coprime.

check reference

check reference

We then have to find parameters $D, v \in \mathbb{Z}$ such that the criteria in 5.32 hold. We get the following:

$$\begin{aligned} 4q &= t^2 + |D|v^2 && \Rightarrow \\ 20 &= (-3)^2 + |D|v^2 && \Leftrightarrow \\ 11 &= |D|v^2 \end{aligned}$$

Now, since 11 is a prime number, the only solution is $|D| = 11$ and $v = 1$ here. With $D = -11$ and the Euclidean division of -11 by 4 being $-11 = -3 \cdot 4 + 1$, we have $-11 \bmod 4 = 1$, which shows that $D = -11$ is a proper choice.

In the next step, we have to compute the Hilbert class polynomial H_{-11}. To do so, we first have to find the set $ICG(D)$. To compute that set, observe that, since $\sqrt{\frac{|D|}{3}} \approx 1.915 < 2$, we know from $A \leq \sqrt{\frac{|D|}{3}}$ and $A \in \mathbb{Z}$ that A must be either 0 or 1.

For $A = 0$, we know $B = 0$ from the constraint $|B| \leq A$. However, in this case, there could be no C satisfying $-11 = B^2 - 4AC$. So we try $A = 1$ and deduce $B \in \{-1, 0, 1\}$ from the constraint $|B| \leq A$. The case $B = -1$ can be excluded, since then $B < 0$ has to imply $|B| < A$. The case $B = 0$ can also be excluded, as there cannot be an integer C with $-11 = -4C$, since 11 is a prime number.

This leaves the case $B = 1$, and we compute $C = 3$ from the equation $-11 = 1^2 - 4C$, which gives the solution $(A, B, C) = (1, 1, 3)$:

$$ICG(D) = \{(1,1,3)\}$$

With the set $ICG(D)$ at hand, we can compute the Hilbert class polynomial of $D = -11$. To do so, we have to insert the term $\frac{-1+\sqrt{-11}}{2\cdot 1}$ into the j-function. To do so, first observe that $\sqrt{-11} = i\sqrt{11}$, where i is the imaginary unit, defined by $i^2 = -1$. Using this, we can invoke Sage to compute the j-invariant and get the following:

$$H_{-11}(x) = x - j\left(\frac{-1+i\sqrt{11}}{2}\right) = x + 32768$$

As we can see, in this particular case, the Hilbert class polynomial is a linear function with a single integer coefficient. In the next step, we have to project it onto a polynomial from $\mathbb{F}_5[x]$ by computing the modular 5 remainder of the coefficients 1 and 32768. We get $32768 \bmod 5 = 3$, from which it follows that the projected Hilbert class polynomial is considered a polynomial from $\mathbb{F}_5[x]$:

$$H_{-11,5}(x) = x + 3$$

As we can see, the only root of this polynomial is $j = 2$, since $H_{-11,5}(2) = 2 + 3 = 0$. We therefore have a situation with $j \neq 0$ and $j \neq 1728$, which tells us that we have to compute the parameter c_1 in modular 5 arithmetics:

$$c_1 = \frac{2}{1728 - 2}$$

Since $1728 \bmod 5 = 3$, we get $c_1 = 2$.

Next, we have to check if the curve $E(\mathbb{F}_5)$ defined by the Weierstraß equation $y^2 = x^3 + 3 \cdot 2x + 2 \cdot 2$ has the correct order. We invoke Sage, and find that the order is indeed 9, so it is a curve with the required parameters. Thus, we have successfully constructed the curve with the desired properties.

Note, however, that in real-world applications, it might be useful to choose parameters a and b that have certain properties, e.g. to be a small as possible. As we know from XXX, choosing any quadratic residue $d \in \mathbb{F}_5$ gives a curve of the same order defined by $y^2 = x^2 + ak^2x + bk^3$. Since 4 is a quadratic residue in $\mathbb{F}_4$, we can transform the curve defined by $y^2 = x^3 + x + 4$ into the curve $y^2 = x^3 + 4^2 + 4 \cdot 4^3$ which gives the following:

add reference

$$y^2 = x^3 + x + 1$$

This is the curve $E_1(\mathbb{F}_5)$ that we used extensively throughout this book. Thus, using the complex multiplication method, we were able to derive a curve with specific properties from scratch.

Example 101. Consider the tiny-jubjub curve *TJJ_13* from example 66. We want to use the complex multiplication method to derive that curve from scratch. Since *TJJ_13* is a curve of order $r = 20$ over the prime field of order $q = 13$, we know from example 95 that its trace of Frobenius is $t = -6$, which also implies that q and $|t|$ are coprime.

check reference

check reference

We then have to find parameters $D, v \in \mathbb{Z}$ such that 5.32 holds. We get the following:

$$\begin{aligned} 4q &= t^2 + |D|v^2 && \Rightarrow \\ 4 \cdot 13 &= (-6)^2 + |D|v^2 && \Rightarrow \\ 52 &= 36 + |D|v^2 && \Leftrightarrow \\ 16 &= |D|v^2 \end{aligned}$$

This equation has two solutions for (D, v), namely $(-4, \pm 2)$ and $(-16, \pm 1)$. Looking at the first solution, we know that $D = -4$ implies $j = 1728$, and the constructed curve is defined by a Weierstraß equation 5.1 that has a vanishing parameter $b = 0$. We can therefore conclude that choosing $D = -4$ will not help us reconstructing *TJJ_13*. It will produce curves with order 20, just not the one we are looking for.

check reference

So we choose the second solution $D = -16$. In the next step, we have to compute the Hilbert class polynomial H_{-16}. To do so, we first have to find the set $ICG(D)$. To compute that set, observe that since $\sqrt{\frac{|-16|}{3}} \approx 2.31 < 3$, we know from $A \leq \sqrt{\frac{|-16|}{3}}$ and $A \in \mathbb{Z}$ that A must be in the range 0..2. So we loop through all possible values of A and through all possible values of B under the constraints $|B| \leq A$, and if $B < 0$ then $|B| < A$. Then we compute potential C's from $-16 = B^2 - 4AC$. We get the following two solutions for $ICG(D)$: we get

$$ICG(D) = \{(1,0,4),(2,0,2)\}$$

With the set $ICG(D)$ at hand, we can compute the Hilbert class polynomial of $D = -16$. We can invoke Sage to compute the j-invariant and get the following:

$$\begin{aligned} H_{-16}(x) &= \left(x - j\left(\frac{i\sqrt{16}}{2}\right)\right)\left(x - j\left(\frac{i\sqrt{16}}{4}\right)\right) \\ &= (x - 287496)(x - 1728) \end{aligned}$$

As we can see, in this particular case, the Hilbert class polynomial is a quadratic function with two integer coefficients. In the next step, we have to project it onto a polynomial from $\mathbb{F}_5[x]$ by computing the modular 5 remainder of the coefficients 1, 287496 and 1728. We get $287496 \bmod 13 = 1$ and $1728 \bmod 13 = 2$, which means that the projected Hilbert class polynomial is as follows:

$$H_{-11,5}(x) = (x-1)(x-12) = (x+12)(x+1)$$

This is considered a polynomial from $\mathbb{F}_5[x]$. Thus, we have two roots, namely $j = 1$ and $j = 12$. We already know that $j = 12$ is the wrong root to construct the tiny-jubjub curve, since $1728 \bmod 13 = 2$, and that case is not compatible with a curve with $b \neq 0$. So we choose $j = 1$.

Another way to decide the proper root is to compute the j-invariant of the tiny-jubjub curve. We get the following:

$$\begin{aligned} j(TJJ\_13) &= 12\frac{4\cdot 8^3}{4\cdot 8^3+1\cdot 8^2} \\ &= 12\frac{4\cdot 5}{4\cdot 5+12} \\ &= 12\frac{7}{7+12} \\ &= 12\frac{7}{7+12} \\ &= 1 \end{aligned}$$

This is equal to the root $j=1$ of the Hilbert class polynomial $H_{-16,13}$ as expected. We therefore have a situation with $j\neq 0$ and $j\neq 1728$, which tells us that we have to compute the parameter c_1 in modular 5 arithmetics:

$$c_1 = \frac{1}{12-1} = 6$$

Since 1728 mod 13 $=12$, we get $c_1=6$. Then we have to check if the curve $E(\mathbb{F}_5)$ defined by the Weierstraß equation $y^2=x^3+3\cdot 6x+2\cdot 6$, which is equivalent to $y^2=x^3+5x+12$, has the correct order. We invoke Sage and find that the order is 8, which implies that the trace of this curve is 6, not -6 as required. So we have to consider the second possibility, and choose some quadratic non-residue $c_2\in\mathbb{F}_{13}$. We choose $c_2=5$ and compute the Weierstraß equation $y^2=x^3+5c_2^2+12c_2^3$ as follows:

$$y^2=x^3+8x+5$$

We invoke Sage and find that the order is 20, which is indeed the correct one. As we know from XXX, choosing any quadratic residue $d\in\mathbb{F}_5$ gives a curve of the same order defined by add reference $y^2=x^2+ad^2x+bd^3$. Since 12 is a quadratic residue in $\mathbb{F}_{13}$, we can transform the curve defined by $y^2=x^3+8x+5$ into the curve $y^2=x^3+12^2\cdot 8+5\cdot 12^3$ which gives the following:

$$y^2=x^3+8x+8$$

This is the tiny-jubjub curve that we used extensively throughout this book. So using the complex multiplication method, we were able to derive a curve with specific properties from scratch.

Example 102. To consider a real-world example, we want to use the complex multiplication method in combination with Sage to compute Secp256k1 from scratch. So based on example 67, we decided to compute an elliptic curve over a prime field $\mathbb{F}_p$ of order r for the following check reference security parameters:

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$
$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

According to example 96, this gives the following trace of Frobenius: check reference

$$t = 432420386565659656852420866390673177327$$

We also decided that we want a curve of the form $y^2=x^3+b$, that is, we want the parameter a to be zero. This implies that the j-invariant of our curve must be zero.

In a first step, we have to find a CM-discriminant D and some integer v such that the equation $4p = t^2 + |D|v^2$ is satisfied. Since we aim for a vanishing j-invariant, the first thing to try is $D = -3$. In this case, we can compute $v^2 = (4p - t^2)$, and if v^2 happens to be an integer that has a square root v, we are done. Invoking Sage we compute as follows:

```
sage: D = -3
sage: p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
sage: r = 115792089237316195423570985008687907852837564279074904382605163141518161494337
sage: t = p+1-r
sage: v_sqr = (4*p - t^2)/abs(D)
sage: v_sqr.is_integer()
True
sage: v = sqrt(v_sqr)
sage: v.is_integer()
True
sage: 4*p == t^2 + abs(D)*v^2
True
sage: v
303414439467246543595250775667605759171
```

The pair $(D, v) = (-3, 303414439467246543595250775667605759171)$ does indeed solve the equation, which tells us that there is a curve of order r over a prime field of order p, defined by a Weierstraß equation $y^2 = x^3 + b$ for some $b \in \mathbb{F}_p$. Now we need to compute b.

For $D = -3$, we already know that the associated Hilbert class polynomial is given by $H_{-3}(x) = x$, which gives the projected Hilbert class polynomial as $H_{-3,p} = x$ and the j-invariant of our curve is guaranteed to be $j = 0$. Now, looking at 5.4.0.1, we see that there are 6 possible cases to construct a curve with the correct order r. In order to construct the curves in question, we have to choose some arbitrary quadratic and cubic non-residue. So we loop through $\mathbb{F}_p$ to find them, invoking Sage:

check reference

```
sage: F = GF(p)
sage: for c2 in F:
....:     try: # quadratic residue
....:         _ = c2.nth_root(2)
....:     except ValueError: # quadratic non-residue
....:         break
sage: c2
3
sage: for c3 in F:
....:     try:
....:         _ = c3.nth_root(3)
....:     except ValueError:
....:         break
sage: c3
2
```

We found the quadratic non-residue $c_2 = 3$ and the cubic non-residue $c_3 = 2$. Using those numbers, we check the six cases against the the expected order r of the curve we want to synthesize:

```
sage: C1 = EllipticCurve(F,[0,1])
sage: C1.order() == r
False
sage: C2 = EllipticCurve(F,[0,c2^3])
sage: C2.order() == r
False
sage: C3 = EllipticCurve(F,[0,c3^2])
sage: C3.order() == r
False
sage: C4 = EllipticCurve(F,[0,c3^2*c2^3])
sage: C4.order() == r
False
sage: C5 = EllipticCurve(F,[0,c3^(-2)])
sage: C5.order() == r
False
sage: C6 = EllipticCurve(F,[0,c3^(-2)*c2^3])
sage: C6.order() == r
True
```

As expected, we found an elliptic curve of the correct order r over a prime field of size p. In principle. we are done, as we have found a curve with the same basic properties as Secp256k1. However, the curve is defined by the following equation, which uses a very large parameter b_1, and so it might perform too slowly in certain algorithms.

$$y^2 = x^3 + 86844066927987146567678238756515930889952488499230423029593188005931626003754$$

It is also not very elegant to be written down by hand. It might therefore be advantageous to find an isomorphic curve with the smallest possible parameter b_2. In order to find such a b_2, we have to choose a quadratic residue d such that $b_2 = b_1 \cdot d^3$ is as small as possible. To do so, we rewrite the last equation into the following form:

what does this mean? Maybe just delete it

$$d = \sqrt[3]{\frac{b_2}{b_1}}$$

Then we invoke Sage to loop through values $b_2 \in \mathbb{F}_p$ until it finds some number such that the quotient $\frac{b_2}{b_1}$ has a cube root d and this cube root itself is a quadratic residue.

```
sage: b1=8684406692798714656767823875651593088995248849923042302959318800593162600 3754
sage: for b2 in F:
....:     try:
....:         d = (b2/b1).nth_root(3)
....:         try:
....:             _ = d.nth_root(2)
....:             if d != 0:
....:                 break
....:         except ValueError:
....:             pass
....:     except ValueError:
....:         pass
```

```
sage: b2
7
```

Indeed, the smallest possible value is $b_2 = 7$ and the defining Weierstraß equation of a curve over $\mathbb{F}_p$ with prime order r is $y^2 = x^3 + 7$, which we might call Secp256k1. As we have just seen, the complex multiplication method is powerful enough to derive cryptographically secure curves like Secp256k1 from scratch.

The ***BLS*****6_6 pen-and-paper curve** In this paragraph, we summarize our understanding of elliptic curves to derive our main pen-and-paper example for the rest of the book. To do so, we want to use the complex multiplication method to derive a pairing-friendly elliptic curve that has similar properties to curves that are used in actual cryptographic protocols. However, we design the curve specifically to be useful in pen-and-paper examples, which mostly means that the curve should contain only a few points so that we are able to derive exhaustive addition and pairing tables.

A well-understood family of pairing-friendly curves is the the group of BLS curves (STUFF ABOUT THE HISTORY AND THE NAMING CONVENTION), which are derived in [XXX]. BLS curves are particularly useful in our case if the embedding degree k satisfies $k \equiv 6 \pmod 0$. Of course, the smallest embedding degree k that satisfies this congruency is $k = 6$ and we therefore aim for a BLS6 curve as our main pen-and-paper example.

write up this part

add reference

To apply the complex multiplication method from page 111 ff., recall that this method starts with a definition of the base field $\mathbb{F}_{p^m}$, as well as the trace of Frobenius t and the order of the curve. If the order $p^m + 1 - t$ is not a prime number, then the order r of the largest prime factor group needs to be controlled.

check reference

In the case of BLS_6 curves, the parameter m is chosen to be 1, which means that the curves are defined over prime fields. All relevant parameters p, t and r are then themselves parameterized by the following three polynomials:

$$
\begin{aligned}
r(x) &= \Phi_6(x) \\
t(x) &= x + 1 \\
q(x) &= \frac{1}{3}(x-1)^2(x^2 - x + 1) + x
\end{aligned}
\tag{5.38}
$$

In the equations above, Φ_6 is the 6-th cyclotomic polynomial and $x \in \mathbb{N}$ is a parameter that the designer has to choose in such a way that the evaluation of p, t and r at the point x gives integers that have the proper size to meet the security requirements of the curve that they want to design. It is then guaranteed that the complex multiplication method can be used in combination with those parameters to define an elliptic curve with CM-discriminant $D = -3$, embedding degree $k = 6$, and curve equation $y^2 = x^3 + b$ for some $b \in \mathbb{F}_p$.

cyclotomic polynomial

For example, if the curve should target the 128-bit security level, due to the Pholaard-rho attack (TODO) the parameter r should be prime number of at least 256 bits.

Pholaard-rho attack

todo

In order to design the smallest BLS_6 curve, we therefore have to find a parameter x such that $r(x)$, $t(x)$ and $q(x)$ are the smallest natural numbers that satisfy $q(x) > 3$ and $r(x) > 3$.[1]

We therefore initiate the design process of our *BLS*6 curve by looking up the 6-th cyclotomic polynomial, which is $\Phi_6 = x^2 - x + 1$, and then insert small values for x into the defining

[1]The smallest BLS curve will also be the most insecure BLS curve. However, since our goal with this curve is ease of pen-and-paper computation rather than security, it fits the purposes of this book.

polynomials r,t,q. We get the following results:

$$\begin{array}{lll} x=1 & (r(x),t(x),q(x)) & (1,2,1) \\ x=2 & (r(x),t(x),q(x)) & (3,3,3) \\ x=3 & (r(x),t(x),q(x)) & (7,4,\frac{37}{3}) \\ x=4 & (r(x),t(x),q(x)) & (13,5,43) \end{array}$$

Since $q(1)=1$ is not a prime number, the first x that gives a proper curve is $x=2$. However, such a curve would be defined over a base field of characteristic 3, and we would rather like to avoid that. We therefore find $x=4$, which defines a curve over the prime field of characteristic 43 that has a trace of Frobenius $t=5$ and a larger order prime group of size $r=13$.

why? Because in this book elliptic curves are only defined for fields of chracteristic > 3

Since the prime field $\mathbb{F}_{43}$ has 43 elements and 43's binary representation is $43_2 = 101011$, which consists of 6 digits, the name of our pen-and-paper curve should be $BLS6\_6$, since its is common to name a BLS curve by its embedding degree and the bit-length of the modulus in the base field. We call $BLS6\_6$ the **moon-math-curve**.

Based on 5.29, we know that the Hasse bound implies that $BLS6\_6$ will contain exactly 39 elements. Since the prime factorization of 39 is $39 = 3 \cdot 13$, we have a "large" prime factor group of size 13, as expected, and a small cofactor group of size 3. Fortunately, a subgroup of order 13 is well suited for our purposes, as 13 elements can be easily handled in the associated addition, scalar multiplication and pairing tables in a pen-and-paper style.

check reference

We can check that the embedding degree is indeed 6 as expected, since $k=6$ is the smallest number k such that $r=13$ divides 43^k-1.

```
sage: for k in range(1,42): # Fermat's little theorem
....:     if (43^k-1)%13 == 0:
....:         break
sage: k
6
```

In order to compute the defining equation $y^2 = x^3 + ax + b$ of BLS6-6, we use the complex multiplication method as described in 5.4. The goal is to find $a,b \in \mathbb{F}_{43}$ representations that are particularly nice to work with. The authors of XXX showed that the CM-discriminant of every BLS curve is $D=-3$ and, indeed, the following equation has the four solutions $(D,v) \in \{(-3,-7),(-3,7),(-49,-1),(-49,1)\}$ if D is required to be negative, as expected:

check reference

what does this mean?

add reference

$$\begin{aligned} 4p &= t^2 + |D|v^2 & \Rightarrow \\ 4\cdot 43 &= 5^2 + |D|v^2 & \Rightarrow \\ 172 &= 25 + |D|v^2 & \Leftrightarrow \\ 49 &= |D|v^2 \end{aligned}$$

This means that $D=-3$ is indeed a proper CM-discriminant, and we can deduce that the parameter a has to be 0, and that the Hilbert class polynomial is given by $H_{-3,43}(x) = x$.

This implies that the j-invariant of $BLS6\_6$ is given by $j(BLS6\_6) = 0$. We therefore have to look at case XXX in table 5.4.0.1 to derive a parameter b. To decide the proper case for $j_0 = 0$ and $D=-3$, we therefore have to choose some arbitrary quadratic non-residue c_2 and cubic non-residue c_3 in $\mathbb{F}_{43}$. We choose $c_2 = 5$ and $c_3 = 36$. We check these with Sage:

add reference

check reference

```
sage: F43 = GF(43)
```

```
sage: c2 = F43(5)
....: try: # quadratic residue
....:     c2.nth_root(2)
....: except ValueError: # quadratic non-residue
....:     c2
sage: c3 =F43(36)
....: try:
....:     c3.nth_root(3)
....: except ValueError:
....:     c3
```

Using those numbers we check the six possible cases from 5.4.0.1 against the the expected order 39 of the curve we want to synthesize:

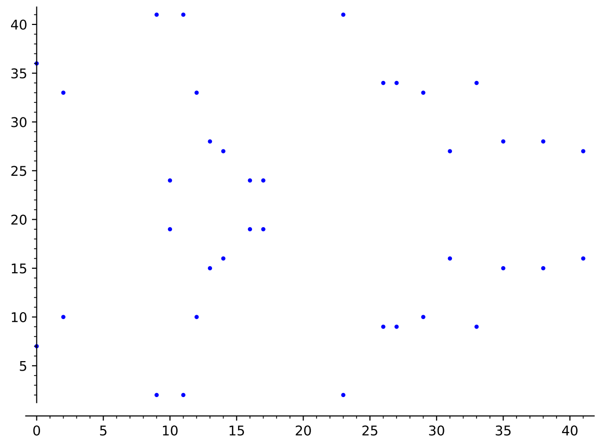
check reference

```
sage: BLS61 = EllipticCurve(F43,[0,1])
sage: BLS61.order() == 39
False
sage: BLS62 = EllipticCurve(F43,[0,c2^3])
sage: BLS62.order() == 39
False
sage: BLS63 = EllipticCurve(F43,[0,c3^2])
sage: BLS63.order() == 39
True
sage: BLS64 = EllipticCurve(F43,[0,c3^2*c2^3])
sage: BLS64.order() == 39
False
sage: BLS65 = EllipticCurve(F43,[0,c3^(-2)])
sage: BLS65.order() == 39
False
sage: BLS66 = EllipticCurve(F43,[0,c3^(-2)*c2^3])
sage: BLS66.order() == 39
False
sage: BLS6 = BLS63 # our BLS6 curve in the book
```

As expected, we found an elliptic curve of the correct order 39 over a prime field of size 43, defined by the following equation:

$$BLS6\_6 := \{(x,y) \mid y^2 = x^3 + 6 \text{ for all } x,y \in \mathbb{F}_{43}\} \tag{5.39}$$

There are other choices for b, such as $b = 10$ or $b = 23$, but all these curves are isomorphic, and hence represent the same curve in a different way. Since BLS6-6 only contains 39 points ,it is possible to give a visual impression of the curve:



As we can see, our curve has some desirable properties: it does not contain self-inverse points, that is, points with $y = 0$. It follows that the addition law can be optimized, since the branch for those cases can be eliminated.

Summarizing the previous procedure, we have used the method of Barreto, Lynn and Scott to construct a pairing-friendly elliptic curve of embedding degree 6. However, in order to do elliptic curve cryptography on this curve, note that, since the order of $BLS6\_6$ is 39, its group of rational points is not a finite cyclic group of prime order. We therefore have to find a suitable subgroup as our main target. Since $39 = 13 \cdot 3$, we know that the curve must contain a "large" prime-order group of size 13 and a small cofactor group of order 3.

add reference

The following step is to construct this group. One way to do so is to find a generator. We can achieve this by choosing an arbitrary element of the group that is not the point at infinity, and then multiply that point with the cofactor of the group's order. If the result is not the point at infinity, the result will be a generator. If it is the point at infinity we have to choose a different element.

In order to find a generator for the large order subgroup of size 13, we first notice that the cofactor of 13 is 3, since $39 = 3 \cdot 13$. We then need to construct an arbitrary element from $BLS6\_6$. To do so in a pen-and-paper style, we can choose some $arbitrary x \in \mathbb{F}_{43}$ and see if there is some solution $y \in \mathbb{F}_{43}$ that satisfies the defining Weierstraß equation $y^2 = x^3 + 6$. We choose $x = 9$, and check that $y = 2$ is a proper solution:

$$
\begin{aligned}
y^2 &= x^3 + 6 && \Rightarrow \\
2^2 &= 9^3 + 6 && \Leftrightarrow \\
4 &= 4
\end{aligned}
$$

This implies that $P = (9,2)$ is therefore a point on $BLS6\_6$. To see if we can project this point onto a generator of the large order prime group $BLS6\_6[13]$, we have to multiply P with the cofactor, that is, we have to compute $[3](9,2)$. After some computation (EXERCISE) we get $[3](9,2) = (13,15)$. Since this is not the point at infinity, we know that $(13,15)$ must be a generator of $BLS6\_6[13]$. The generator $g_{BLS6\_6[13]}$, which we will use in pairing computations in the remainder of this book, is given as follows:

add exercise

$$g_{BLS6\_6[13]} = (13,15) \tag{5.40}$$

Since $g_{BLS6\_6[13]}$ is a generator, we can use it to construct the subgoup $BLS6\_6[13]$ by repeatedly adding the generator to itself. Using Sage, we get the following:

```
sage: P = BLS6(9,2)
```
607

```
sage: Q = 3*P
sage: Q.xy()
(13, 15)
sage: BLS6_13 = []
sage: for x in range(0,13): # cyclic of order 13
....:     P = x*Q
....:     BLS6_13.append(P)
```

Repeatedly adding a generator to itself, as we just did, will generate small groups in logarithmic order with respect to the generator as, explained on page 44 ff. We therefore get the following description of the large prime-order subgroup of $BLS6\_6$: check reference

$$
\begin{aligned}
BLS6\_6[13] = \\
&\{(13,15) \to (33,34) \to (38,15) \to (35,28) \to (26,34) \to (27,34) \to \\
&\quad (27,9) \to (26,9) \to (35,15) \to (38,28) \to (33,9) \to (13,28) \to \mathcal{O}\} \quad (5.41)
\end{aligned}
$$

Having a logarithmic description of this group is tremendously helpful in pen-and-paper computations. To see that, observe that we know fromXXX that there is an exponential map from the scalar field $\mathbb{F}_{13}$ to $BLS6\_6[13]$ with respect to our generator, which generates the group in logarithmic order: add reference

$$[\cdot]_{(13,15)} : \mathbb{F}_{13} \to BLS6\_6[13]\ ;\ x \mapsto [x](13,15)$$

So, for example, we have $[1]_{(13,15)} = (13,15)$, $[7]_{(13,15)} = (27,9)$ and $[0]_{(13,15)} = \mathcal{O}$ and so on. The relevant point here is that we can use this representation to do computations in $BLS6\_6[13]$ efficiently in our head using XXX, as in the following example: add reference

$$
\begin{aligned}
(27,34) \oplus (33,9) &= [6](13,15) \oplus [11](13,15) \\
&= [6+11](13,15) \\
&= [4](13,15) \\
&= (35,28)
\end{aligned}
$$

So XXX is really all we need to do computations in $BLS6\_6[13]$ in this book efficiently. However, out of convenience, the following picture lists the entire addition table of that group, as it might be useful in pen-and-paper computations: add reference

$\oplus$	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)
$\mathcal{O}$	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)
(13,15)	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$
(33,34)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)
(38,15)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)
(35,28)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)
(26,34)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)
(27,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)
(27,9)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)
(26,9)	(26,9)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)
(35,15)	(35,15)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)
(38,28)	(38,28)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)
(33,9)	(33,9)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)
(13,28)	(13,28)	$\mathcal{O}$	(13,15)	(33,34)	(38,15)	(35,28)	(26,34)	(27,34)	(27,9)	(26,9)	(35,15)	(38,28)	(33,9)

Now that we have constructed a "large" cyclic prime-order subgroup of $BLS6\_6$ suitable for many pen-and-paper computations in elliptic curve cryptography, we have to look at how to do pairings in this context. We know that $BLS6\_6$ is a pairing-friendly curve by design, since it has a small embedding degree $k = 6$. It is therefore possible to compute Weil pairings efficiently. However, in order to do so, we have to decide the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ as explained in exercise 73. check reference

Since $BLS6\_6$ has two non-trivial subgroups, it would be possible to use any of them as the n-torsion group. However, in cryptography, the only secure choice is to use the large prime-order subgroup, which in our case is $BLS6\_6[13]$. We therefore decide to consider the 13-torsion and define $G_1[13]$ as the first argument for the Weil pairing function:

$$\mathbb{G}_1[13] = \{(13,15) \to (33,34) \to (38,15) \to (35,28) \to (26,34) \to (27,34) \to \\ (27,9) \to (26,9) \to (35,15) \to (38,28) \to (33,9) \to (13,28) \to \mathcal{O}\}$$

In order to construct the domain for the second argument, we need to construct $\mathbb{G}_2[13]$, which, according to the general theory, should be defined by those elements P of the full 13-torsion group $BLS6\_6[13]$ that are mapped to $43 \cdot P$ under the Frobenius endomorphism (equation 5.24). check reference

To compute $\mathbb{G}_2[13]$, we therefore have to find the full 13-torsion group first. To do so, we use the technique from XXX, which tells us that the full 13-torsion can be found in the curve extension over the extension field $\mathbb{F}_{43^6}$, since the embedding degree of $BLS6\_6$ is 6:

$$BLS6\_6 := \{(x,y) \mid y^2 = x^3 + 6 \text{ for all } x, y \in \mathbb{F}_{43^6}\} \tag{5.42}$$

Thus, we have to construct $\mathbb{F}_{43^6}$, a field that contains 6321363049 elements. In order to do so, we use the procedure of XXX and start by choosing a non-reducible polynomial of degree 6 from the ring of polynomials $\mathbb{F}_{43}[t]$. We choose $p(t) = t^6 + 6$. Using Sage, we get the following: add reference

```
sage: F43 = GF(43)
sage: F43t.<t> = F43[]
sage: p = F43t(t^6+6)
sage: p.is_irreducible()
True
sage: F43_6.<v> = GF(43^6, name='v', modulus=p)
```

Recall from XXX that elements $x \in \mathbb{F}_{43^6}$ can be seen as polynomials $a_0 + a_1 v + a_2 v^2 + \ldots + a_5 v^5$ with the usual addition of polynomials and multiplication modulo $t^6 + 6$. add reference

In order to compute $\mathbb{G}_2[13]$, we first have to extend $BLS6\_6$ to $\mathbb{F}_{43^6}$, that is, we keep the defining equation, but expand the domain from $\mathbb{F}_{43}$ to $\mathbb{F}_{43^6}$. After that, we have to find at least one element P from that curve that is not the point at infinity, is in the full 13-torsion and satisfies the identity $\pi(P) = [43]P$. We can then use this element as our generator of $\mathbb{G}_2[13]$ and construct all other elements by repeatedly adding the generator to itself.

Since $BLS6(\mathbb{F}_{43^6})$ contains 6321251664 elements, it's not a good strategy to simply loop through all elements. Fortunately, Sage has a way to loop through elements from the torsion group directly:

```
sage: BLS6 = EllipticCurve (F43_6,[0 ,6]) # curve extension
sage: INF = BLS6(0) # point at infinity
```

```
sage: for P in INF.division_points(13): # full 13-torsion
....: # PI(P) == [q]P
....:     if P.order() == 13: # exclude point at infinity
....:         PiP = BLS6([a.frobenius() for a in P])
....:         qP = 43*P
....:         if PiP == qP:
....:             break
sage: P.xy()
(7*v^2, 16*v^3)
```

We found an element from the full 13-torsion that is in the Eigenspace of the Eigenvalue 43, which implies that it is an element of $\mathbb{G}_2[13]$. As $\mathbb{G}_2[13]$ is cyclic of prime order, this element must be a generator:

$$g_{\mathbb{G}_2[13]} = (7v^2, 16v^3) \tag{5.43}$$

We can use this generator to compute $\mathbb{G}_2$ in logarithmic order with respect to $g_{\mathbb{G}[13]}$. Using Sage we get the following:

```
sage: Q = BLS6(7*v^2,16*v^3)
sage: BLS6_13_2 = []
sage: for x in range(0,13):
....:     P = x*Q
....:     BLS6_13_2.append(P)
```

$$\begin{aligned}\mathbb{G}_2 = \{&(7v^2,16v^3) \to (10v^2,28v^3) \to (42v^2,16v^3) \to (37v^2,27v^3) \to \\ &(16v^2,28v^3) \to (17v^2,28v^3) \to (17v^2,15v^3) \to (16v^2,15v^3) \to \\ &(37v^2,16v^3) \to (42v^2,27v^3) \to (10v^2,15v^3) \to (7v^2,27v^3) \to \mathcal{O}\}\end{aligned}$$

Again, having a logarithmic description of $\mathbb{G}_2[13]$ is tremendously helpful in pen-and-paper computations, as it reduces complicated computation in the extended curves to modular 13 arithmetics, as in the following example:

$$\begin{aligned}(17v^2,28v^3) \oplus (10v^2,15v^2) &= [6](7v^2,16v^3) \oplus [11](7v^2,16v^3) \\ &= [6+11](7v^2,16v^3) \\ &= [4](7v^2,16v^3) \\ &= (37v^2,27v^3)\end{aligned}$$

So XXX is really all we need to do computations in $\mathbb{G}_2[13]$ in this book efficiently.

add reference

To summarize the previous steps, we have found two subgroups, $\mathbb{G}_1[13]$ and $\mathbb{G}_2[13]$ suitable to do Weil pairings on $BLS6\_6$ as explained in 5.28. Using the logarithmic order XXX of $\mathbb{G}_1[13]$, the logarithmic order XXX of $\mathbb{G}_2[13]$ and the bilinearity in 5.44, we can do Weil pairings on $BLS6\_6$ in a pen-and-paper style:

check reference

add reference

add reference

$$e([k_1]g_{BLS6\_6[13]}, [k_2]g_{\mathbb{G}_2[13]}) = e(g_{BLS6\_6[13]}, g_{\mathbb{G}_2[13]})^{k_1 \cdot k_2} \tag{5.44}$$

Observe that the Weil pairing between our two generators is given by the following identity:

$$e(g_{BLS6\_6[13]}, g_{\mathbb{G}_2[13]}) = 5v^5 + 16v^4 + 16v^3 + 15v^2 + 3v + 41 \tag{5.45}$$

```
sage: g1 = BLS6([13,15])
sage: g2 = BLS6([7*v^2, 16*v^3])
sage: g1.weil_pairing(g2,13)
5*v^5 + 16*v^4 + 16*v^3 + 15*v^2 + 3*v + 41
```

Hashing to pairing groups We give various constructions to hash into $\mathbb{G}_1$ and $\mathbb{G}_2$.

We start with hashing to the scalar field... TO APPEAR

finish writing this up

None of these techniques work for hashing into $\mathbb{G}_2$. We therefore implement Pederson's Hash for BLS6.

We start with $\mathbb{G}_1$. Our goal is to define an 12-bit bounded hash function:

$$H_1 : \{0,1\}^{12} \to \mathbb{G}_1$$

Since $12 = 3 \cdot 4$ we "randomly" select 4 uniformly distributed generators $\{(38,15),(35,28),(27,34),(38,28)\}$ from $\mathbb{G}_1$ and use the pseudo-random function from XXX. Therefore, we have to choose a set of 4 randomly generated invertible elements from $\mathbb{F}_{13}$ for every generator. We choose the following:

add reference

$$\begin{array}{ccc}(38,15) & : & \{2,7,5,9\}\\ (35,28) & : & \{11,4,7,7\}\\ (27,34) & : & \{5,3,7,12\}\\ (38,28) & : & \{6,5,1,8\}\end{array}$$

Our hash function is then computed as follows:

$$\begin{aligned}H_1(x_{11},x_1,\ldots,x_0) = [2\cdot 7^{x_{11}}\cdot 5^{x_{10}}\cdot 9^{x_9}](38,15)+[11\cdot 4^{x_8}\cdot 7^{x_7}\cdot 7^{x_6}](35,28)+\\ [5\cdot 3^{x_5}\cdot 7^{x_4}\cdot 12^{x_3}](27,34)+[6\cdot 5^{x_2}\cdot 1^{x_1}\cdot 8^{x_0}](38,28)\end{aligned}$$

Note that $a^x = 1$ when $x = 0$. Hence, those terms can be omitted in the computation. In particular, the hash of the 12-bit zero string is given as follows:

correct computations

$$WRONG-ORDERING-REDO$$

$$\begin{aligned}H_1(0) = [2](38,15)+[11](35,28)+[5](27,34)+[6](38,28) =\\ (27,34)+(26,34)+(35,28)+(26,9) = (33,9)+(13,28) = (38,28)\end{aligned}$$

The hash of 011010101100 is given as follows:

fill in missing parts

$$\begin{aligned}H_1(011010101100) = WRONG-ORDERING-REDO\\ [2\cdot 7^0\cdot 5^1\cdot 9^1](38,15)+[11\cdot 4^0\cdot 7^1\cdot 7^0](35,28)+[5\cdot 3^1\cdot 7^0\cdot 12^1](27,34)+[6\cdot 5^1\cdot 1^0\cdot 8^0](38,28) =\\ [2\cdot 5\cdot 9](38,15)+[11\cdot 7](35,28)+[5\cdot 3\cdot 12](27,34)+[6\cdot 5](38,28) =\\ [12](38,15)+[12](35,28)+[11](27,34)+[4](38,28) =\\ TOAPPEAR\end{aligned}$$

We can use the same technique to define a 12-bit bounded hash function in $\mathbb{G}_2$:

$$H_2 : \{0,1\}^{12} \to \mathbb{G}_2$$

Again, we "randomly" select 4 uniformly distributed generators $\{(7v^2,16v^3),(42v^2,16v^3),(17v^2,15v^3),(10v^2,15v^3)\}$ from $\mathbb{G}_2$, and use the pseudo-random function from XXX. Therefore, we have to choose a set of 4 randomly generated invertible elements from $\mathbb{F}_{13}$ for every generator:

add reference

$$
\begin{array}{lcl}
(7v^2,16v^3) & : & \{8,4,5,7\} \\
(42v^2,16v^3) & : & \{12,1,3,8\} \\
(17v^2,15v^3) & : & \{2,3,9,11\} \\
(10v^2,15v^3) & : & \{3,6,9,10\}
\end{array}
$$

Our hash function is then computed like this:

$$
\begin{aligned}
H_1(x_{11},x_{10},\ldots,x_0) = {} & [8\cdot 4^{x_{11}}\cdot 5^{x_{10}}\cdot 7^{x_9}](7v^2,16v^3)+[12\cdot 1^{x_8}\cdot 3^{x_7}\cdot 8^{x_6}](42v^2,16v^3)+\\
& [2\cdot 3^{x_5}\cdot 9^{x_4}\cdot 11^{x_3}](17v^2,15v^3)+[3\cdot 6^{x_2}\cdot 9^{x_1}\cdot 10^{x_0}](10v^2,15v^3)
\end{aligned}
$$

We extend this to a hash function that maps unbounded bitstrings to $\mathbb{G}_2$ by precomposing with an actual hash function like $MD5$, and feed the first 12 bits of its outcome into our previously defined hash function, with $TinyMD5_{\mathbb{G}_2}(s) = H_2(MD5(s)_0,\ldots MD5(s)_{11})$:

$$
TinyMD5_{\mathbb{G}_2} : \{0,1\}^* \to \mathbb{G}_2
$$

For example, since $MD5(\text{""}) =$ $0xd41d8cd98f00b204e9800998ecf8427e$, and the binary representation of the hexadecimal number $0x27e$ is 001001111110, we compute $TinyMD5_{\mathbb{G}_2}$ of the empty string as follows:

$$
TinyMD5_{\mathbb{G}_2}(\text{""}) = H_2(MD5(s)_{11},\ldots MD5(s)_0) = H_2(001001111110) =
$$

4097 check equation

Chapter 6

Statements

As we have seen in the informal introduction XXX, a SNARK is a short non-interactive argument of knowledge, where the knowledge-proof attests to the correctness of statements like “The proofer knows the prime factorization of a given number” or “The proofer knows the preimage to a given SHA2 digest value” and similar things. However, human-readable statements like these are imprecise and not very useful from a formal perspective.

Chapter 1?

In this chapter we therefore look more closely at ways to formalize statements in mathematically rigorous ways, useful for SNARK development. We start by introducing formal languages as a way to define statements properly (section 6.1). We will then look at algebraic circuits and rank-1 constraint systems as two particularly useful ways to define statements in certain formal languages (section6.2). After that, we will have a look at fundamental building blocks of compilers that compile high-level languages to circuits and associated rank-1 constraint systems.

Proper statement design should be of high priority in the development of SNARKs, since unintended true statements can lead to potentially severe and almost undetectable security vulnerabilities in the applications of SNARKs.

6.1 Formal Languages

Formal languages provide the theoretical background in which statements can be formulated in a logically regious way and where proofing the correctness of any given statement can be realized by computing words in that language.

"rigorous"?

"proving"?

One might argue that the understanding of formal languages is not very important in SNARK development and associated statement design, but terms from that field of research are standard jargon in many papers on zero-knowledge proofs. We therefore believe that at least some introduction to formal languages and how they fit into the picture of SNARK development is beneficial, mostly to give developers a better intuition about where all this is located in the bigger picture of the logic landscape. In addition, formal languages give a better understanding of what a formal proof for a statement actually is.

Roughly speaking, a formal language (or just language for short) is nothing but a set of words, th. Words, in turn, are strings of letters taken from some alphabet and formed according to some defining rules of the language.

To be more precise, let Σ be any set and Σ^* the set of all finite **tuples** (ordered lists) $(x_1,\ldots,x_n)$ of elements x_j from Σ including the empty tuple $(\ ) \in \Sigma^*$. Then, a **language** L, in its most general definition, is nothing but a subset of Σ^*. In this context, the set Σ is called the **alphabet** of the language L, elements from Σ are called letters and elements from L are called **words**. The rules that specify which tuples from Σ^* belong to the language and which don’t, are called the **grammar** of the language. S: I suggest adding an example based on English, e.g. "tea" and "eat" are words of English, but "aet" and "tae" are not

Add example

If L_1 and L_2 are two formal languages over the same alphabet, we call L_1 and L_2 **equivalent** if they generate the same set of words.

M: 1:1 correspondence might actually be wrong

Decision Functions Our previous definition of formal languages is very general and many subclasses of languages are known in the literature. However, in the context of SNARK development, languages are commonly defined as **decision problems** where a so-called **deciding relation** $R \subset \Sigma^*$ decides whether a given tuple $x \in \Sigma^*$ is a word in the language or not. If $x \in R$ then x is a word in the associated language L_R and if $x \notin R$ then not. The relation R therefore summarizes the grammar of language L_R.

Unfortunately, in some literature on proof systems, $x \in R$ is often written as $R(x)$, which is misleading since in general R is not a function but a relation in Σ^*. For the sake of this book, we therefore adopt a different point of view and work with what we might call a **decision function** instead:

$$R : \Sigma^* \to \{true, false\} \tag{6.1}$$

Decision functions decide if a tuple $x \in \Sigma^*$ is an element of a language or not. In case a decision function is given, the associated language itself can be written as the set of all tuples that are decided by R, i.e as the set:

$$L_R := \{x \in \Sigma^* \mid R(x) = true\} \tag{6.2}$$

In the context of formal languages and decision problems, a **statement** S is the claim that language L contains a word x, i.e a statement claims that there exist some $x \in L$. A constructive **proof** for statement S is given by some string $P \in \Sigma^*$ and such a proof is **verified** by checking $R(P) = true$. In this case, P is called an **instance** of the statement S.

While the term **language** might suggest a deeper relation to the well known **natural languages** like English, formal languages and natural languages differ in many ways. The following examples will provide some intuition about formal languages, highlighting the concepts of statements, proofs and instances:

Example 103 (Alternating Binary strings). To consider a very basic formal language with an almost trivial grammar, consider the set $\{0,1\}$ of the two letters 0 and 1 as our alphabet Σ and imply the rule that a proper word must consist of alternating binary letters of arbitrary length.

Then, the associated language L_{alt} is the set of all finite binary tuples, where a 1 must follow a 0 and vice versa. So, for example, $(1,0,1,0,1,0,1,0,1) \in L_{alt}$ is a proper word in this languages as is $(0) \in L_{alt}$ or the empty word $(\ ) \in L_{alt}$. However, the binary tuple $(1,0,1,0,1,0,1,1,1) \in \{0,1\}^*$ is not a proper word, as it violates the grammar of L_{alt}: the last3 letters are all 1. Furthermore, the tuple $(0,A,0,A,0,A,0)$ is not a proper word, as not all its letters are not from the proper alphabet: we defined the alphabet Σ as the set $\{0,1\}$, and A is not part of that set.

Attempting to write the grammar of this language in a more formal way, we can define the following decision function:

$$R : \{0,1\}^* \to \{true, false\}\ ;\ (x_0, x_1, \ldots, x_n) \mapsto \begin{cases} true & x_{j-1} \neq x_j \text{ for all } 1 \leq j \leq n \\ false & \text{else} \end{cases}$$

We can use this function to decide if arbitrary binary tuples are words in L_{alt} or not. Some examples are given below:

binary tuples

- $R(1,0,1) = true$,

- $R(0) = true$,
- $R() = true$,
- but $R(1,1) = false$.

Inside our language L_{alt}, it makes sense to claim the following statement: "There exists an alternating string." One way to prove this statement constructively is by providing an actual instance, that is, finding actual alternating string like $x = (1,0,1)$. Constructing string $(1,0,1)$ therefore proves the statement "There exists an alternating string.", because it is easy to verify that $R(1,0,1) = true$.

Example 104 (Programming Language). Programming languages are a very important class of formal languages. For these languages, the alphabet is usually (a subset) of the ASCII table, and the grammar is defined by the rules of the programming language's compiler. Words, then, are nothing but properly written computer programs that the compiler accepts. The compiler can therefore be interpreted as the decision function.

To give an unusual example strange enough to highlight the point, consider the programming language Malbolge as defined in XXX. This language was specifically designed to be almost impossible to use and writing programs in this language is a difficult task. An interesting claim is therefore the statement: "There exists a computer program in Malbolge". As it turned out, proving this statement constructively, that is, by providing an actual instance of such a program, was not an easy task, as it took two years after the introduction of Malbolge to write a program that its compiler accepts. So, for two years, no one was able to prove the statement constructively.

add reference

To look at this high-level description more formally, we write $L_{Malbolge}$ for the language that uses the ASCII table as its alphabet and its words are tuples of ASCII letters that the Malbolge compiler accepts. Proving the statement "There exists a computer program in Malbolge" is then equivalent to the task of finding some word $x \in L_{Malbolge}$. The string

(=<'#9] 6ZY327Uv4–QsqpMn&+Ij"'E%e{Ab w=_:]Kw%o44Uqp0/Q?xNvL:'H%c#DD2∧WV>gY;dts76qKJImZkj

is an example of such a proof, as it is excepted by the Malbolge compiler and is compiled to an executable binary that displays "Hello, World." (See XXX). In this example, the Malbolge compiler therefore serves as the verification process.

add reference

Example 105 (The Empty Language). To see that not every language has even one word, consider the alphabet $\Sigma = \mathbb{Z}_6$, where $\mathbb{Z}_6$ is the ring of modular 6 arithmetics as derived in example 8 in chapter 3, together with the following decision function

check reference

$$R_\emptyset : \mathbb{Z}_6^* \to \{true, false\} \; ; \; (x_1, \ldots, x_n) \mapsto \begin{cases} true & n = 4 \text{ and } x_1 \cdot x_1 = 2 \\ true & else \end{cases}$$

We write $L_\emptyset$ for the associated language. As we can see from the multiplication table of $\mathbb{Z}_6$ in example 8 in chapter 3, the ring $\mathbb{Z}_6$ does not contain any element x such that $x^2 = 2$, which implies $R_\emptyset(x_1, \ldots, x_n) = false$ for all tuples $(x_1, \ldots, x_n) \in \Sigma^*$. The language therefore does not contain any words. Proving the statement "There exists a word in $L_\emptyset$" constructively by providing an instance is therefore impossible. The verification will never check any tuple.

check reference

Example 106 (3-Factorization). We will use the following simple example repeatedly throughout this book. The task is to develop a SNARK that proves knowledge of three factors of an element from the finite field $\mathbb{F}_{13}$. There is nothing particularly useful about this example from an application point of view, however, in a sense, it is the most simple example that gives rise to a non trivial SNARK in some of the most common zero-knowledge proof systems.

Formalizing the high-level description, we use $\Sigma := \mathbb{F}_{13}$ as the underlying alphabet of this problem and define the language $L_{3.fac}$ to consists of those tuples of field elements from $\mathbb{F}_{13}$ that contain exactly 4 letters w_1, w_2, w_3, w_4 which satisfy the equation $w_1 \cdot w_2 \cdot w_3 = w_4$.

Are we using w and x interchangeably or is there a difference between them?

So, for example, the tuple $(2,12,4,5)$ is a word in $L_{3.fac}$, while neither $(2,12,11)$, nor $(2,12,4,7)$ nor $(2,12,7,168)$ are words in $L_{3.fac}$ as they don't satisfy the grammar or are not define over the proper alphabet.

We can describe the language $L_{3.fac}$ more formally by introducing a decision function (as described in equation 6.1):

check reference

$$R_{3.fac} : \mathbb{F}_{13}^* \to \{true, false\} \; ; \; (x_1,\ldots,x_n) \mapsto \begin{cases} true & n = 4 \text{ and } x_1 \cdot x_2 \cdot x_3 = x_4 \\ false & else \end{cases}$$

Having defined the language $L_{3.fac}$, it then makes sense to claim the statement "There is a word in $L_{3.fac}$". The way $L_{3.fac}$ is designed, this statement is equivalent to the statement "There are four elements w_1, w_2, w_3, w_4 from the finite field $\mathbb{F}_{13}$ such that the equation $w_1 \cdot w_2 \cdot w_3 = w_4$ holds."

Proving the correctness of this statement constructively means to actually find some concrete field elements like $x_1 = 2$, $x_2 = 12$, $x_3 = 4$ and $x_4 = 5$ that satisfy the relation $R_{3.fac}$. The tuple $(2,12,4,5)$ is therefore a constructive proof for the statement and the computation $R_{3.fac}(2,12,4,5) = true$ is a verification of that proof. In contrast, the tuple $(2,12,4,7)$ is not a proof of the statement, since the check $R_{3.fac}(2,12,4,7) = false$ does not verify the proof.

Example 107 (Tiny-jubjub Membership). In one of our main examples, we derive a SNARK that proves a pair (x,y) of field elements from $\mathbb{F}_{13}$ to be a point on the tiny-jubjub curve in its **Edwards form** (see section 5.1.3).

jubjub

check reference

In the first step, we define a language such that points on the tiny-jubjub curve are in 1:1 correspondence with words in that language.

Since the tiny-jubjub curve is an elliptic curve over the field $\mathbb{F}_{13}$, we choose the alphabet $\Sigma = \mathbb{F}_{13}$. In this case, the set $\mathbb{F}_{13}^*$ consists of all finite strings of field elements from $\mathbb{F}_{13}$. To define the grammar, recall from 66 that a point on the tiny-jubjub curve is a pair (x,y) of field elements such that $3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$. We can use this equation to derive the following decision function:

check reference

$$R_{tiny.jj} : \mathbb{F}_{13}^* \to \{true, false\} \; ; \; (x_1,\ldots,x_n) \mapsto \begin{cases} true & n = 2 \text{ and } 3 \cdot x_1^2 + x_2^2 = 1 + 8 \cdot x_1^2 \cdot x_2^2 \\ false & else \end{cases}$$

The associated language $L_{tiny.jj}$ is then given as the set of all strings from $\mathbb{F}_{13}^*$ that are mapped onto $true$ by $R_{tiny.jj}$. We get

$$L_{tiny.jj} = \{(x_1,\ldots,x_n) \in \mathbb{F}_{13}^* \mid R_{tiny.jj(x_1,\ldots,x_n)=true}\}$$

We can claim the statement "There is a word in $L_{tiny.jj}$" and because $L_{tiny.jj}$ is defined by $R_{tiny.jj}$, this statement is equivalent to the claim "The tiny-jubjub curve in its Edwards form has curve a point."

check wording

A constructive proof for this statement is a pair (x,y) of field elements that satisfies the Edwards equation. Example 66 therefore implies that the tuple $(11,6)$ is a constructive proof and the computation $R_{tiny.jj}(11,6) = true$ is a proof verification. In contrast, the tuple $(1,1)$ is not a proof of the statement, since the check $R_{tiny.jj}(1,1) = false$ does not verify the proof.

check reference

Exercise 43. Consider exercise XXX again. Define a decision function such that the associated language $L_{Exercise_XXX}$ consist precisely of all solutions to the equation $5x+4=28+2x$ over $\mathbb{F}_{13}$. Provide a constructive proof for the claim: "There exist a word in $L_{Exercise_XXX}$ and verify the proof.

Exercise 44. Consider the modular 6 arithmetics $\mathbb{Z}_6$ from example 8 in chapter 3, the alphabet $\Sigma = \mathbb{Z}_6$ and the decision function

$$R_{example\_8} : \Sigma^* \to \{true, false\} \; ; \; x \mapsto \begin{cases} true & x.len() = 1 \text{ and } 3\cdot x + 3 = 0 \\ false & else \end{cases}$$

Compute all words in the associated language $L_{example\_8}$, provide a constructive proof for the statement "There exist a word in $L_{example\_example\_8}$" and verify the proof.

check references

Instance and Witness As we have seen in the previous paragraph, statements provide membership claims in formal languages, and instances serve as constructive proofs for those claims. However, in the context of **zero-knowledge** proof systems, our naive notion of constructive proofs is refined in such a way that its possible to hide parts of the proof instance and still be able to prove the statement. In this context, it is therefore necessary to split a proof into a **public part** called the **instance** and a private part called a **witness**.

To account for this separation of a proof instance into a public and a private part, our previous definition of formal languages needs a refinement in the context of zero-knowledge proof systems. Instead of a single alphabet, the refined definition considers two alphabets Σ_I and Σ_W, and a decision function defined as follows:

$$R : \Sigma_I^* \times \Sigma_W^* \to \{true, false\} \; ; \; (i;w) \mapsto R(i;w) \tag{6.3}$$

Words are therefore tuples $(i;w) \in \Sigma_I^* \times \Sigma_W^*$ with $R(i;w) = true$. The refined definition differentiates between public inputs $i \in \Sigma_I$ and private inputs $w \in \Sigma_W$. The public input i is called an **instance** and the private input w is called a **witness** of R.

If a decision function is given, the associated language is defined as the set of all tuples from the underlying alphabet that are verified by the decision function:

$$L_R := \{(i;w) \in \Sigma_I^* \times \Sigma_W^* \mid R(i;w) = true\} \tag{6.4}$$

In this refined context, a **statement** S is a claim that, given an instance $i \in \Sigma_I^*$, there is a witness $w \in \Sigma_W^*$ such that language L contains a word $(i;w)$. A constructive **proof** for statement S is given by some string $P = (i;w) \in \Sigma_I^* \times \Sigma_W^*$ and a proof is **verified** by checking $R(P) = true$.

It is worth understanding the difference between statements as defined in XXX and the refined notion of statements from this paragraph. While statements in the sense of the previous paragraph can be seen as membership claims, statements in the refined definition can be seen as knowledge-proofs, where a prover claims knowledge of a witness for a given instance. For a more detailed discussion on this topic see [XXX sec 1.4]

add reference

add reference

Example 108 (SHA256 – Knowlege of Preimage). One of the most common examples in the context of zero-knowledge proof systems is the **knowledge-of-a-preimage proof** for some cryptographic hash function like $SHA256$, where a publicly known $SHA256$ digest value is given, and the task is to prove knowledge of a preimage for that digest under the $SHA256$ function, without revealing that preimage.

To understand this problem in detail, we have to introduce a language able to describe the knowledge-of-preimage problem in such a way that the claim "Given digest i, there is a preimage w such that $SHA256(w) = i$" becomes a statement in that language. Since $SHA256$ is a function

$$SHA256 : \{0,1\}^* \to \{0,1\}^{256}$$

that maps binary strings of arbitrary length onto binary strings of length 256 and we want to prove knowledge of preimages, we have to consider binary strings of size 256 as instances and binary strings of arbitrary length as witnesses.

An appropriate alphabet Σ_I for the set of all instances and an appropriate alphabet Σ_W for the set of all witnesses is therefore given by the set $\{0,1\}$ of the two binary letters and a proper decision function is given by:

$$R_{SHA256} : \{0,1\}^* \times \{0,1\}^* \to \{true, false\} ;$$

$$(i;w) \mapsto \begin{cases} true & i.len() = 256,\ i = SHA256(w) \\ false & else \end{cases}$$

We write L_{SHA256} for the associated language and note that it consists of words, which are tuples $(i;w)$ such that the instance i is the $SHA256$ image of the witness w.

Given some instance $i \in \{0,1\}^{256}$, a statement in L_{SHA256} is the claim "Given digest i, there is a preimage w such that $SHA256(w) = i$", which is exactly what the knowledge-of-preimage problem is about. A constructive proof for this statement is therefore given by a preimage w to the digest i and proof verification is achieved by checking that $SHA256(w) = i$.

Example 109 (3-factorization). To give an intuition about the implication of refined languages, consider $L_{3.fac}$ from example 106 again. As we have seen, a constructive proof in $L_{3.fac}$ is given by 4 field elements x_1, x_2, x_3 and x_4 from $\mathbb{F}_{13}$ such that the product in modular 13 arithmetics of the first three elements is equal to the 4'th element.

check reference

Splitting words from $L_{3.fac}$ into private and public parts, we can reformulate the problem and introduce different levels of privacy into the problem. For example, we could reformulate the membership statement of $L_{3.fac}$ into a statement where all factors x_1, x_2, x_3 of x_4 are private and only the product x_4 is public. A statement for this reformulation is then expressed by the claim: "Given a publicly known field element x_4, there are three private factors of x_4". Assuming some instance x_4, a constructive proof for the associated knowledge claim is then provided by any tuple (x_1, x_2, x_3) such that $x_1 \cdot x_2 \cdot x_3 = x_4$.

At this point, it is important to note that, while constructive proofs in the refinement don't look very different from constructive proofs in the original language, we will see in XXX that there are proof systems able to prove the statement (at least with high probability) without revealing anything about the factors x_1, x_2, or x_3. This is why the importance of the refinement only becomes clear once more elaborate proofing methods beyond naive constructive proofs are provided.

add reference

We can formalize this new language, which we might call $L_{3.fac\_zk}$, by defining the following decision function:

Definition 6.1.0.1.

$$R_{3.fac\_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{true, false\} ;$$

$$((i_1, \ldots, i_n);(w_1, \ldots, w_m)) \mapsto \begin{cases} true & n = 1,\ m = 3,\ i_1 = w_1 \cdot w_2 \cdot w_3 \\ false & else \end{cases}$$

The associated language $L_{3.fac\_zk}$ is defined by all tuples from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto $true$ under the decision function $R_{3.fac\_zk}$.

Considering the distinction we made between the instance and the witness part in $L_{3.fac\_zk}$, one might ask why we chose the factors x_1, x_2 and x_3 to be the witness and the product x_4 to be the instance and why we didn't choose another combination? This was an arbitrary choice in the example. Every other combination of private and public factors would be equally valid. For example, it would be possible to declare all variables as private or to declare all variables as public. Actual choices are determined by the application only.

Example 110 (The Tiny-Jubjub Curve). Consider the language $L_{tiny.jj}$ from example 107. As we have seen, a constructive proof in $L_{tiny.jj}$ is given by a pair (x_1,x_2) of field elements from $\mathbb{F}_{13}$ such that the pair is a point of the tiny-jubjub curve in its Edwards representation. check reference

We look at a reasonable splitting of words from $L_{tiny.jj}$ into private and public parts. The two obvious choices are to either choose both coordinates x_1 as x_2 as public inputs, or to choose both coordinates x_1 as x_2 as private inputs.

In case both coordinates are public, we define the grammar of the associated language by introducing the following decision function:

$$R_{tiny.jj.1} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{true, false\} ;$$
$$(I_1,\dots,I_n;W_1,\dots,W_m) \mapsto \begin{cases} true & n=2, m=0 \text{ and } 3\cdot I_1^2 + I_2^2 = 1 + 8\cdot I_1^2\cdot I_2^2 \\ false & else \end{cases}$$

The language $L_{tiny.jj.1}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto $true$ by $R_{tiny.jj.1}$.

In case both coordinates are private, we define the grammar of the associated refined language by introducing the following decision function:

$$R_{tiny.jj\_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{true, false\} ;$$
$$(I_1,\dots,I_n;W_1,\dots,W_m) \mapsto \begin{cases} true & n=0, m=m \text{ and } 3\cdot W_1^2 + W_2^2 = 1 + 8\cdot W_1^2\cdot W_2^2 \\ false & else \end{cases}$$

The language $L_{tiny.jj\_zk}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto $true$ by $R_{tiny.jj\_zk}$.

Exercise 45. Consider the modular 6 arithmetics $\mathbb{Z}_6$ from example 8 in chapter 3 as alphabets Σ_I and Σ_W and the following decision function check reference

$$R_{linear} : \Sigma^* \times \Sigma^* \to \{true, false\} ;$$
$$(i;w) \mapsto \begin{cases} true & i.len() = 3 \text{ and } w.len() = 1 \text{ and } i_1\cdot w_1 + i_2 = i_3 \\ false & else \end{cases}$$

Which of the following instances (i_1,i_2,i_3) has a proof of knowledge in L_{linear}?

- $(3,3,0)$
- $(2,1,0)$
- $(4,4,2)$

Exercise 46 (Edwards Addition on Tiny-Jubjub). Consider the tiny-jubjub curve together with its Edwards addition law from example XXX. Define an instance alphabet Σ_I, a witness alphabet Σ_W and a decision function R_{add} with associated language L_{add} such that a string $(i;w) \in \Sigma_I^* \times$ add reference

Σ_W^* is a word in L_{add} if and only if i is a pair of curve points on the tiny-jubjub curve in Edwards form and w is the Edwards sum of those curve points.

Choose some instance $i \in \Sigma_I^*$, provide a constructive proof for the statement "There is a witness $w \in \Sigma_W^*$ such that $(i;w)$ is a word in L_{add}" and verify that proof. Then find some instance $i \in \Sigma_I^*$ such that i has no knowledge proof in L_{add}.

Modularity From a developers perspective, it is often useful to construct complex statements and their representing languages from simple ones. In the context of zero-knowledge proof systems, those simple building blocks are often called **gadgets**, and gadget libraries usually contain representations of atomic types like booleans, integers, various hash functions, elliptic curve cryptography and many more. In order to synthesize statements, developers then combine predefined gadgets into complex logic. We call the ability to combine statements into more complex statements **modularity**.

To understand the concept of modularity on the level of formal languages defined by decision functions, we need to look at the **intersection** of two languages, which exists whenever both languages are defined over the same alphabet. In this case, the intersection is a language that consists of strings which are words in both languages.

To be more precise, let L_1 and L_2 be two languages defined over the same instance and witness alphabets Σ_I and Σ_W. Then the intersection $L_1 \cap L_2$ of L_1 and L_2 is defined as

$$L_1 \cap L_2 := \{x \mid x \in L_1 \text{ and } x \in L_2\} \tag{6.5}$$

If both languages are defined by decision functions R_1 and R_2, the following function is a decision function for the intersection language $L_1 \cap L_2$:

$$R_{L_1 \cap L_2} : \Sigma_I^* \times \Sigma_W^* \to \{true, false\} \; ; \; (i,w) \mapsto R_1(i,w) \text{ and } R_2(i,w) \tag{6.6}$$

Thus, the intersection of two decision-function-based languages is a also decision-function-based language. This is important from an implementations point of view: It allows us to construct complex decision functions, their languages and associated statements from simple building blocks. Given a publicly known instance $i \in \Sigma_I^*$ a statement in an intersection language then claims knowledge of a witness that satisfies all relations simultaneously.

6.2 Statement Representations

As we have seen in the previous section, formal languages and their definitions by decision functions are a powerful tool to describe statements in a formally rigorous manner.

However, from the perspective of existing zero-knowledge proof systems, not all ways to actually represent decision functions are equally useful. Depending on the proof system, some are more suitable than others. In this section, will describe two of the most common ways to represent decision functions and their statements.

6.2.1 Rank-1 Quadratic Constraint Systems

Although decision functions are expressible in various ways, many contemporary proofing systems require the deciding relation to be expressed in terms of a system of quadratic equations over a finite field. This is true in particular for pairing-based proofing systems like XXX, roughly because it is possible to check solutions to those equations "in the exponent" of pairing-friendly cryptographic groups.

add reference

In this section, we will therefore have a closer look at a particular type of quadratic equation called **rank-1 quadratic constraints systems**, which are a common standard in zero-knowledge proof systems. We will start with a general introduction to those systems and then look at their relation to formal languages. We will look into a common way to compute solutions to those systems, and then show how a simple compiler might derive rank-1 constraint systems from more high-level programming code.

R1CS representation To understand what **rank-1 (quadratic) constraint systems** are in detail, let $\mathbb{F}$ be a field, n, m and $k \in \mathbb{N}$ three numbers and a^i_j, b^i_j and $c^i_j \in \mathbb{F}$ constants from $\mathbb{F}$ for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$. Then a rank-1 constraint system (R1CS) is defined as follows:

Definition 6.2.1.1. **R1CS representation**

$$
\begin{gathered}
\left(a_0^1+\sum_{j=1}^n a_j^1\cdot I_j+\sum_{j=1}^m a_{n+j}^1\cdot W_j\right)\cdot\left(b_0^1+\sum_{j=1}^n b_j^1\cdot I_j+\sum_{j=1}^m b_{n+j}^1\cdot W_j\right)=c_0^1+\sum_{j=1}^n c_j^1\cdot I_j+\sum_{j=1}^m c_{n+j}^1\cdot W_j\\
\vdots\\
\left(a_0^k+\sum_{j=1}^n a_j^k\cdot I_j+\sum_{j=1}^m a_{n+j}^k\cdot W_j\right)\cdot\left(b_0^k+\sum_{j=1}^n b_j^k\cdot I_j+\sum_{j=1}^m b_{n+j}^k\cdot W_j\right)=c_0^k+\sum_{j=1}^n c_j^k\cdot I_j+\sum_{j=1}^m c_{n+j}^k\cdot W_j
\end{gathered}
$$

If a rank-1 constraint system is given, the parameter k is called the **number of constraints** If a tuple $(I_1,\ldots,I_n;W_1,\ldots,W_m)$ of field elements satisfies theses equations, $(I_1,\ldots,I_n)$ is called an **instance** and $(W_1,\ldots,W_m)$ is called an associated **witness** of the system.

Remark 2 (Matrix notation). The presentation of rank-1 constraint systems can be simplified using the notation of vectors and matrices, which abstracts over the indices. In fact if $x = (1,I,W) \in \mathbb{F}^{1+n+m}$ is a $(n+m+1)$-dimensional vector, A, B, C are $(n+m+1)\times k$-dimensional matrices and $\odot$ is the Schur/Hadamard product, then a R1CS can be written as Schur/Hadamard product

$$
Ax \odot Bx = Cx
$$

However, since we did not introduced matrix calculus in the book, we use XXX as the defining equation for rank-1 constraints systems. We only highlighted the matrix notation, because it is sometimes used in the literature. add reference

Generally speaking, the idea of a rank-1 constraint system is to keep track of all the values that any variable can assume during a computation and to bind the relationships among all those variables that are implied by the computation itself. Enforcing relations between all the steps of a computer program, the execution is then constrained to be computed in exactly the expected way without any opportunity for deviations. In this sense, solutions to rank-1 constraint systems are proofs of proper progam execution.

Example 111 (3-Factorization). To provide a better intuition of rank-1 constraint systems, consider the language $L_{3.fac\_zk}$ from example 106 again. As we have seen, $L_{3.fac\_zk}$ consists of words $(I_1;W_1,W_2,W_3)$ over the alphabet $\mathbb{F}_{13}$ such that $I_1 = W_1\cdot W_2\cdot W_3$. We show how to rewrite the decision function as a rank-1 constraint system. check reference

Since R1CS are systems of quadratic equations, expressions like $W_1\cdot W_2\cdot W_3$ which contain products of more than two factors (which are therefore not quadratic) have to be rewritten in a process often called **flattening**. To flatten the defining equation $I_1 = W_1\cdot W_2\cdot W_3$ of $L_{3.fac\_zk}$ we introcuce a new variable W_4, which captures two of the three multiplications in $W_1\cdot W_2\cdot W_3$. We get the following two constraints

$$
\begin{aligned}
W_1\cdot W_2 &= W_4 && \text{constraint 1}\\
W_4\cdot W_3 &= I_1 && \text{constraint 2}
\end{aligned}
$$

Given some instance I_1, any solution (W_1, W_2, W_3, W_4) to this system of equations provides a solution to the original equation $I_1 = W_1 \cdot W_2 \cdot W_3$ and vice versa. Both equations are therefore equivalent in the sense that solutions are in a 1:1 correspondence.

Looking at both equations, we see how each constraint enforces a step in the computation. In fact, the first constraint forces any computation to multiply the witness W_1 and W_2 first. Otherwise it would not be possible to compute the witness W_4, which is needed to solve the second constraint. Witness W_4 therefore expresses the constraining of an intermediate computational state.

At this point, one might ask why equation 1 constrains the system to compute $W_1 \cdot W_2$ first, since computing $W_2 \cdot W_3$, or $W_1 \cdot W_3$ in the beginning and then multiplying with the remaining factor gives the exact same result. The reason is that the way we designed the R1CS prohibits any of these alternative computations, which shows that R1CS are in general **not unique** descriptions of a language: many different R1CS are able to describe the same problem.

To see that the two quadratic equations qualify as a rank-1 constraint system, choose the parameter $n = 1$, $m = 4$ and $k = 2$ as well as

$$\begin{array}{llllll}
a_0^1 = 0 & a_1^1 = 0 & a_2^1 = 1 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\
a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 0 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 1 \\
\\
b_0^1 = 0 & b_1^1 = 0 & b_2^1 = 0 & b_3^1 = 1 & b_4^1 = 0 & b_5^1 = 0 \\
b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 0 & b_3^2 = 0 & b_4^2 = 1 & b_5^2 = 0 \\
\\
c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 0 & c_4^1 = 0 & c_5^1 = 1 \\
c_0^2 = 0 & c_1^2 = 1 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 0 & c_5^2 = 0
\end{array}$$

With this choice, the rank-1 constraint system of our 3-factorization problem can be written in its most general form as follows:

$$\begin{aligned}
&\left(a_0^1+a_1^1 I_1+a_2^1 W_1+a_3^1 W_2+a_4^1 W_3+a_5^1 W_4\right)\cdot\left(b_0^1+b_1^1 I_1+b_2^1 W_1+b_3^1 W_2+b_4^1 W_3+b_5^1 W_4\right)=\left(c_0^1+c_1^1 I_1+c_2^1 W_1+c_3^1 W_2+c_4^1 W_3+c_5^1 W_4\right)\\
&\left(a_0^2+a_1^2 I_1+a_2^2 W_2+a_3^2 W_2+a_4^2 W_3+a_5^2 W_4\right)\cdot\left(b_0^2+b_1^2 I_1+b_2^2 W_2+b_3^2 W_2+b_4^2 W_3+b_5^2 W_4\right)=\left(c_0^2+c_1^2 I_1+c_2^2 W_2+c_3^2 W_2+c_4^2 W_3+c_5^2 W_4\right)
\end{aligned}$$

Example 112 (The Tiny-Jubjub curve). Consider the languages $L_{tiny.jj.1}$ from example 107, check reference
which consist of words (I_1, I_2) over the alphabet $\mathbb{F}_{13}$ such that $3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2$.

We derive a rank-1 constraint system such that its associated language is equivalent to $L_{tiny.jj.1}$. To achieve this, we first rewrite the defining equation:

$$\begin{aligned}
3 \cdot I_1^2 + I_2^2 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 && \Leftrightarrow \\
0 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 - 3 \cdot I_1^2 - I_2^2 && \Leftrightarrow \\
0 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 + 10 \cdot I_1^2 + 12 \cdot I_2^2
\end{aligned}$$

Since R1CSs are systems of quadratic equations, we have to reformulate this expression into a system of quadratic equations. To do so, we have to introduce new variables that constrain intermediate steps in the computation and we have to decide if those variables should be public or private. We decide to declare all new variables as private and get the following constraints

$$\begin{aligned}
I_1 \cdot I_1 &= W_1 && \text{constraint 1} \\
I_2 \cdot I_2 &= W_2 && \text{constraint 2} \\
(8 \cdot W_1) \cdot W_2 &= W_3 && \text{constraint 3} \\
(12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 &= 0 && \text{constraint 4}
\end{aligned}$$

To see that these four quadratic equations qualify as a rank-1 constraint system according to definition XXX, choose the parameter $n = 2$, $m = 3$ and $k = 4$:

add reference

$$\begin{array}{llllll}
a_0^1 = 0 & a_1^1 = 1 & a_2^1 = 0 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\
a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 1 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 0 \\
a_0^3 = 0 & a_1^3 = 0 & a_2^3 = 0 & a_3^3 = 8 & a_4^3 = 0 & a_5^3 = 0 \\
a_0^4 = 1 & a_1^4 = 0 & a_2^4 = 0 & a_3^4 = 10 & a_4^4 = 12 & a_5^4 = 1
\end{array}$$

$$\begin{array}{llllll}
b_0^1 = 0 & b_1^1 = 1 & b_2^1 = 0 & b_3^1 = 0 & b_4^1 = 0 & b_5^1 = 0 \\
b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 1 & b_3^2 = 0 & b_4^2 = 0 & b_5^2 = 0 \\
b_0^3 = 0 & b_1^3 = 0 & b_2^3 = 0 & b_3^3 = 0 & b_4^3 = 1 & b_5^3 = 0 \\
b_0^4 = 1 & b_1^4 = 0 & b_2^4 = 0 & b_3^4 = 0 & b_4^4 = 0 & b_5^4 = 0
\end{array}$$

$$\begin{array}{llllll}
c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 1 & c_4^1 = 0 & c_5^1 = 0 \\
c_0^2 = 0 & c_1^2 = 0 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 1 & c_5^2 = 0 \\
c_0^3 = 0 & c_1^3 = 0 & c_2^3 = 0 & c_3^3 = 0 & c_4^3 = 0 & c_5^3 = 1 \\
c_0^4 = 0 & c_1^4 = 0 & c_2^4 = 0 & c_3^4 = 0 & c_4^4 = 0 & c_5^4 = 0
\end{array}$$

With this choice, the rank-1 constraint system of our tiny-jubjub curve point problem can be written in its most general form as follows:

$$\begin{aligned}
\left(a_0^1+a_1^1I_1+a_2^1I_2+a_3^1W_1+a_4^1W_2+a_5^1W_3\right)\cdot\left(b_0^1+b_1^1I_1+b_2^1I_2+b_3^1W_1+b_4^1W_2+b_5^1W_3\right) &= \left(c_0^1+c_1^1I_1+c_2^1I_2+c_3^1W_1+c_4^1W_2+c_5^1W_3\right)\\
\left(a_0^2+a_1^2I_1+a_2^2I_2+a_3^2W_1+a_4^2W_2+a_5^2W_3\right)\cdot\left(b_0^2+b_1^2I_1+b_2^2I_2+b_3^2W_1+b_4^2W_2+b_5^2W_3\right) &= \left(c_0^2+c_1^2I_1+c_2^2I_2+c_3^2W_1+c_4^2W_2+c_5^2W_3\right)\\
\left(a_0^3+a_1^3I_1+a_2^3I_2+a_3^3W_1+a_4^3W_2+a_5^3W_3\right)\cdot\left(b_0^3+b_1^3I_1+b_2^3I_2+b_3^3W_1+b_4^3W_2+b_5^3W_3\right) &= \left(c_0^3+c_1^3I_1+c_2^3I_2+c_3^3W_1+c_4^3W_2+c_5^3W_3\right)\\
\left(a_0^4+a_1^4I_1+a_2^4I_2+a_3^4W_1+a_4^4W_2+a_5^4W_3\right)\cdot\left(b_0^4+b_1^4I_1+b_2^4I_2+b_3^4W_1+b_4^4W_2+b_5^4W_3\right) &= \left(c_0^4+c_1^4I_1+c_2^4I_2+c_3^4W_1+c_4^4W_2+c_5^4W_3\right)
\end{aligned}$$

In what follows, we write L_{jubjub} for the associated language that consists of solutions to the R1CS.

To see that L_{jubjub} is equivalent to $L_{tiny.jj.1}$, let $(I_1, I_2; W_1, W_2, W_3)$ be a word in L_{jubjub}, then (I_1, I_2) is a word in $L_{tiny.jj.1}$, since the defining R1CS of L_{jubjub} implies that I_1 and I_2 satisfy the Edwards equation of the tiny-jubjub curve. On the other hand, let (I_1, I_2) be a word in $L_{tiny.jj.1}$. Then $(I_1, I_2; I_1^2, I_2^2, 8 \cdot I_1^2 \cdot I_2^2)$ is a word in L_{jubjub} and both maps are inverses of each other.

Exercise 47. Consider the language $L_{tiny.jj\_zk}$ and define a rank-1 constraint relation with a decision function such that the associated language is equivalent to $L_{tiny.jj\_zk}$.

R1CS Satisfiability To understand how rank-1 constraint systems define formal languages, observe that every R1CS over a field $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{R1CS} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\} \; ; \; (I;W) \mapsto \begin{cases} true & (I;W) \text{ satisfies R1CS} \\ false & else \end{cases} \tag{6.7}$$

Every R1CS therefore defines a formal language. The grammar of this language is encoded in the constraints, words are solutions to the equations and a **statement** is a knowledge claim "Given instance I, there is a witness W such that $(I;W)$ is a solution to the rank-1 constraint system". A constructive proof to this claim is therefore an assignment of a field element to every witness variable, which is verified whenever the set of all instance and witness variables solves the R1CS.

Remark 3 (R1CS satisfiability). It should be noted that in our definition, every R1CS defines its own language. However, in more theoretical approaches, another language usually called **R1CS satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness or the computational complexity of the class of **all** R1CS. From our perspective, the R1CS satisfiability language is obtained by the union of all R1CS languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

$$L_{R1CS\_SAT(\mathbb{F})} = \{(i;w) \in \Sigma^* \times \Sigma^* \mid \text{there is a R1CS } R \text{ such that } R(i;w) = true\}$$

Example 113 (3-Factorization). Consider the language $L_{3.fac\_zk}$ from example 106 and the R1CS defined in example ex:3-factorization-r1cs. As we have seen in ex:3-factorization-r1cs, solutions to the R1CS are in 1:1 correspondence with solutions to the decision function of $L_{3.fac\_zk}$. Both languages are therefore equivalent in the sense that there is a 1:1 correspondence between words in both languages.

check reference

check reference

check reference

To give an intuition of what constructive proofs in $L_{3.fac\_zk}$ look like, consider the instance $I_1 = 11$. To prove the statement "There exists a witness W such that $(I_1;W)$ is a word in $L_{3.fac\_zk}$" constructively, a proof has to provide assignments to all witness variables W_1, W_2, W_3 and W_4. Since the alphabet is $\mathbb{F}_{13}$, an example assignment is given by $W = (2,3,4,6)$ since $(I_1;W)$ satisfies the R1CS

$$\begin{aligned} W_1 \cdot W_2 &= W_4 && \# \, 2 \cdot 3 = 6 \\ W_4 \cdot W_3 &= I_1 && \# \, 6 \cdot 4 = 11 \end{aligned}$$

A proper constructive proof is therefore given by $P = (2,3,4,6)$. Of course, P is not the only possible proof for this statement. Since factorization is not unique in a field in general, another constructive proof is given by $P' = (3,5,12,2)$.

Example 114 (The tiny-jubjub curve). Consider the language L_{jubjub} from example 107 and its associated R1CS. To see how constructive proofs in L_{jubjub} look like, consider the instance $(I_1,I_2) = (11,6)$. To prove the statement "There exists a witness W such that $(I_1,I_2;W)$ is a word in L_{jubjub}" constructively, a proof has to provide assignments to all witness variables W_1, W_2 and W_3. Since the alphabet is $\mathbb{F}_{13}$, an example assignment is given by $W = (4,10,8)$ since $(I_1,I_2;W)$ satisfies the R1CS

check reference

$$\begin{aligned} I_1 \cdot I_1 &= W_1 & 11 \cdot 11 &= 4 \\ I_2 \cdot I_2 &= W_2 & 6 \cdot 6 &= 10 \\ (8 \cdot W_1) \cdot W_2 &= W_3 & (8 \cdot 4) \cdot 10 &= 8 \\ (12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 &= 0 & 12 \cdot 10 + 8 + 10 \cdot 4 + 1 &= 0 \end{aligned}$$

A proper constructive proof is therefore given by $P = (4,10,8)$, which shows that the instance $(11,6)$ is a point on the tiny-jubjub curve.

Modularity As we discussed on page 136 XXX, it is often useful to construct complex statements and their representing languages from simple ones. Rank-1 constraint systems are particularly useful for this, as the intersection of two R1CS over the same alphabet results in a new R1CS over that same alphabet.

check reference

To be more precise, let S_1 and S_2 be two R1CS over $\mathbb{F}$, then a new R1CS S_3 is obtained by the intersection $S_3 = S_1 \cap S_2$ of S_1 and S_2. In this context, intersection means that both the equations of S_1 **and** the equations of S_2 have to be satisfied in order to provide a solution for the system S_3.

As a consequence, developers are able to construct complex R1CS from simple ones and this modularity provides the theoretical foundation for many R1CS compilers, as we will see in XXX.

add reference

6.2.2 Algebraic Circuits

As we have seen in the previous paragraphs, rank-1 constraint systems are quadratic equations such that solutions are knowledge proofs for the existence of words in associated languages. From the perspective of a proofer, it is therefore important to solve those equations efficiently.

However, in contrast to systems of linear equation, no general methods are known that solve systems of quadratic equations efficiently. Rank-1 constraint systems are therefore impractical from a proofers perspective and auxiliary information is needed that helps to compute solutions efficiently.

Methods which compute R1CS solutions are sometimes called **witness generator functions**. To provide a common example, we introduce another class of decision functions called **algebraic circuits**. As we will see, every algebraic circuit defines an associated R1CS and also provides an efficient way to compute solutions for that R1CS.

It can be shown that every space- and time-bounded computation is expressible as an algebraic circuit. Transforming high-level computer programs into those circuits is a process often called **flattening**.

To understand this in more detail, we will introduce our model for algebraic circuits and look at the concept of circuit execution and valid assignments. After that, we will show how to derive rank-1 constraint systems from circuits and how circuits are useful to compute solutions to their R1CS efficiently.

Algebraic circuit representation To see what algebraic circuits are, let $\mathbb{F}$ be a field. An algebraic circuit is then a directed acyclic (multi)graph that computes a polynomial function over $\mathbb{F}$. Nodes with only outgoing edges (source nodes) represent the variables and constants of the function and nodes with only incoming edges (sink nodes) represent the outcome of the function. All other nodes have exactly two incoming edges and represent the defining field operations **addition** as well as **multiplication**. Graph edges represent the flow of the computation along the nodes.

To be more precise, we call a directed acyclic multi-graph $C(\mathbb{F})$ an **algebraic circuit** over $\mathbb{F}$ in this book if the following conditions hold:

Definition 6.2.2.1. **Algebraic circuit**

- The set of edges has a total order.
- Every source node has a label that represents either a variable or a constant from the field $\mathbb{F}$.
- Every sink node has exactly one incoming edge and a label that represents either a variable or a constant from the field $\mathbb{F}$.
- Every node that is neither a source nor a sink has exactly two incoming edges and a label from the set $\{+,*\}$ that represents either addition or multiplication in $\mathbb{F}$.
- All outgoing edges from a node have the same label.
- Outgoing edges from a node with a label that represents a variable have a label.

- Outgoing edges from a node with a label that represents multiplication have a label, if there is at least one labeled edge in both input path.
- All incoming edges to sink nodes have a label.
- If an edge has two labels S_i and S_j it gets a new label $S_i = S_j$.
- No other edge has a label.
- Incoming edges to sink nodes that are labeled with a constant $c \in \mathbb{F}$ are labeled with the same constant. Every other edge label is taken from the set $\{W, I\}$ and indexed compatible with the order of the edge set.

It should be noted that the details in the definitions of algebraic circuits vary between different sources. We use this definition as it is conceptually straightforward and well-suited for pen-and-paper computations.

To get a better intuition of our definition, let $C(\mathbb{F})$ be an algebraic circuit. Source nodes are the inputs to the circuit and either represent variables or constants. In a similar way, sink nodes represent termination points of the circuit and are either output variables or constants. Constant sink nodes enforce computational outputs to take on certain values.

Nodes that are neither source nodes nor sink nodes are called **arithmetic gates**. Arithmetic gates that are decorated with the "$+$"-label are called **addition-gates** and arithmetic gates that are decorated with the "$\cdot$"-label are called **multiplication-gates**. Every arithmetic gate has exactly two inputs, represented by the two incoming edges.

Since the set of edges is ordered, we can write it as $\{E_1, E_2, \ldots, E_n\}$ for some $n \in \mathbb{N}$ and we use those indices to index the edge labels, too. Edge labels are therefore either constants or symbols like I_j, W_j or S_j, where j is an index compatible with the edge order. Labels I_j represent instance variables, labels W_j witness variables. Labels on the outgoing edges of input variables constrain the associated variable to that edge. Every other edge defines a constraining equation in the associated R1CS. We will explain this in more detail in XXX. add reference

Notation and Symbols 10. In synthesizing algebraic circuits, assigning instance I_j or witness W_j labels to appropriate edges is often the final step. It is therefore convenient to not distinguish these two types of edges in previous steps. To account for that, we often simply write S_j for an edge label, indicating that the private/public property of the label is unspecified and it might represent an instance or a witness label.

Example 115 (Generalized factorization SNARK). To give a simple example of an algebraic circuit, consider our 3-factorization problem from example 106 again. To express the problem check reference in the algebraic circuit model, consider the following function

$$f_{3.fac} : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 \cdot x_3$$

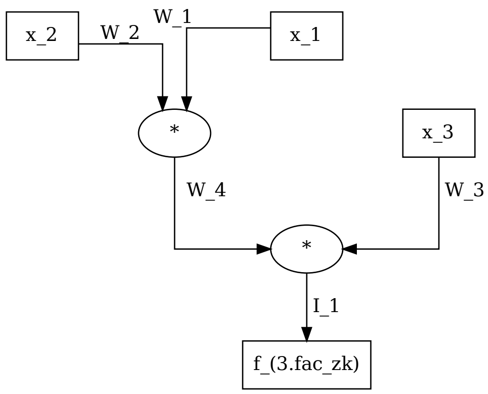
Using this function, we can describe the zero-knowledge 3-factorization problem from 106, check reference in the following way: Given instance $I_1 \in \mathbb{F}_{13}$, a valid witness is a preimage of $f_{3.fac}$ at the point I_1, i.e., a valid witness consists of three values W_1, W_2 and W_3 from $\mathbb{F}_{13}$ such that $f_{3.fac}(W_1, W_2, W_3) = I_1$.

To see how this function can be transformed into an algebraic circuit over $\mathbb{F}_{13}$, it is a common first step to introduce brackets into the function's definition and then write the operations as binary operators, in order to highlight how exactly every field operation acts on its two inputs.

Due to the associativity laws in a field, we have several choices. We choose

$$
\begin{aligned}
f_{3.fac}(x_1,x_2,x_3) &= x_1 \cdot x_2 \cdot x_3 && \text{\# bracket choice}\\
&= (x_1 \cdot x_2)\cdot x_3 && \text{\# operator notation}\\
&= MUL(MUL(x_1,x_2),x_3)
\end{aligned}
$$

Using this expression, we can write an associated algebraic circuit by first constraining the variables to edge labels $W_1 = x_1$, $W_2 = x_2$ and $W_3 = x_3$ as well as $I_1 = f_{3.fac}(x_1,x_2,x_3)$, taking the distinction between private and public inputs into account. We then rewrite the operator representation of $f_{3.fac}$ into circuit nodes and get the following:



In this case, the directed acyclic multi-graph is a binary tree with three leaves (the source nodes) labeled by x_1, x_2 and x_3, one root (the single sink node) labeled by $f(x_1,x_2,x_3)$ and two internal nodes, which are labeled as multiplication gates.

The order we use to label the edges is chosen to make the edge labeling consistent with the choice of W_4 as defined in definition 6.2.2.1. This order can be obtained by a depth-first right-to-left-first traversal algorithm. check reference

Example 116. To give a more realistic example of an algebraic circuit, look at the defining equation of the tiny-jubjub curve (66) again. A pair of field elements $(x,y) \in \mathbb{F}_{13}^2$ is a curve point, precisely if the following equation holds: check reference

$$3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$$

To understand how one might transform this identity into an algebraic circuit, we first rewrite this equation by shifting all terms to the right. We get the following:

$$
\begin{aligned}
3 \cdot x^2 + y^2 &= 1 + 8 \cdot x^2 \cdot y^2 && \Leftrightarrow\\
0 &= 1 + 8 \cdot x^2 \cdot y^2 - 3 \cdot x^2 - y^2 && \Leftrightarrow\\
0 &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2
\end{aligned}
$$

Then we use this expression to define a function such that all points of the tiny-jubjub curve are characterized as the function preimages at 0.

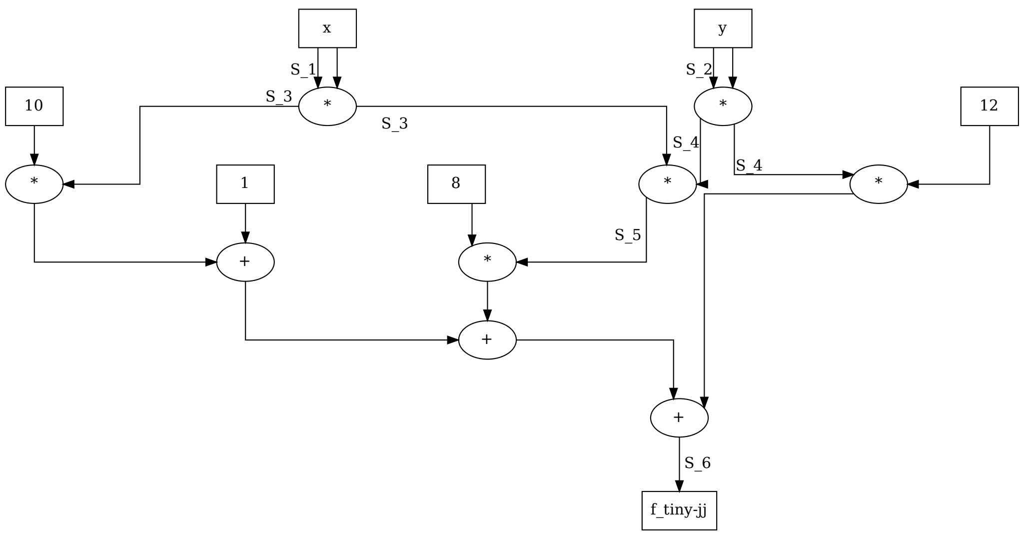
$$f_{tiny-jj} : \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13} \; ; \; (x,y) \mapsto 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2$$

Every pair of points $(x,y) \in \mathbb{F}_{13}^2$ with $f_{tiny-jj}(x,y) = 0$ is a point on the tiny-jubjub curve, and there are no other curve points. The preimage $f_{tiny-jj}^{-1}(0)$ is therefore a complete description of the tiny-jubjub curve.

We can transform this function into an algebraic circuit over $\mathbb{F}_{13}$. We first introduce brackets into potentially ambiguous expressions and then rewrite the function in terms of binary operators. We get the following:

$$
\begin{aligned}
f_{tiny-jj}(x,y) &= 1 + 8\cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12y^2 && \Leftrightarrow\\
&= ((8\cdot((x\cdot x)\cdot(y\cdot y))) + (1 + 10\cdot(x\cdot x))) + (12\cdot(y\cdot y)) && \Leftrightarrow\\
&= ADD(ADD(MUL(8, MUL(MUL(x,x), MUL(y,y))), ADD(1, MUL(10, MUL(x,x)))), MUL(12, MUL(y,y)))
\end{aligned}
$$

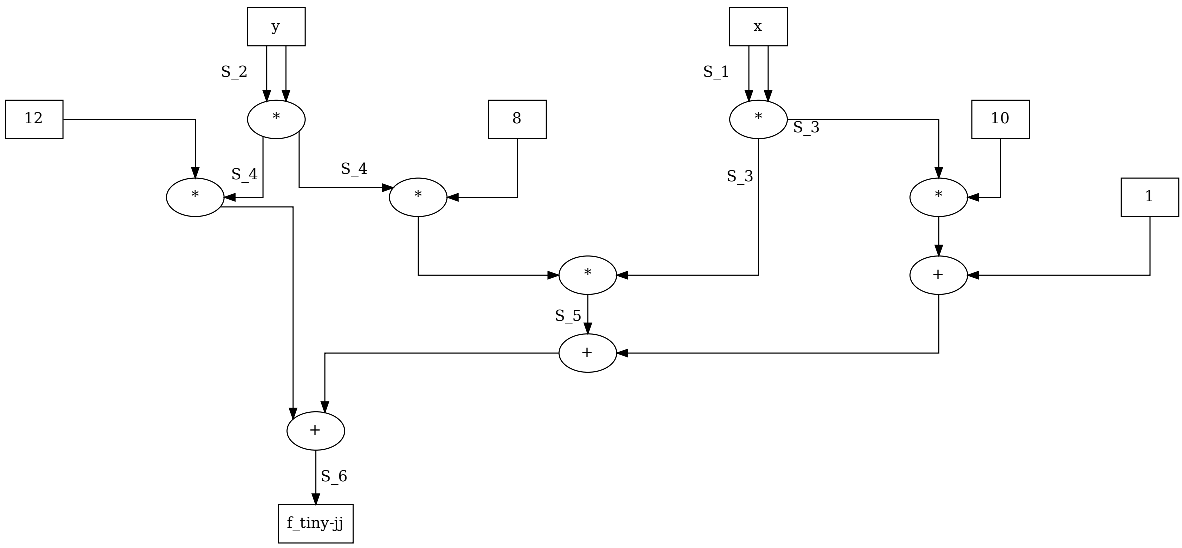
Since we haven't decided which part of the computation should be public and which part should be private, we use the unspecified symbol S to represent edge labels. Constraining all variables to edge labels $S_1 = x$, $S_2 = y$ and $S_6 = f_{tiny-jj}$, we get the following circuit, representing the function $f_{tiny-jj}$, by inductively replacing binary operators with their associated arithmetic gates:



This circuit is not a graph, but a multigraph, since there is more than one edge between some of the nodes.

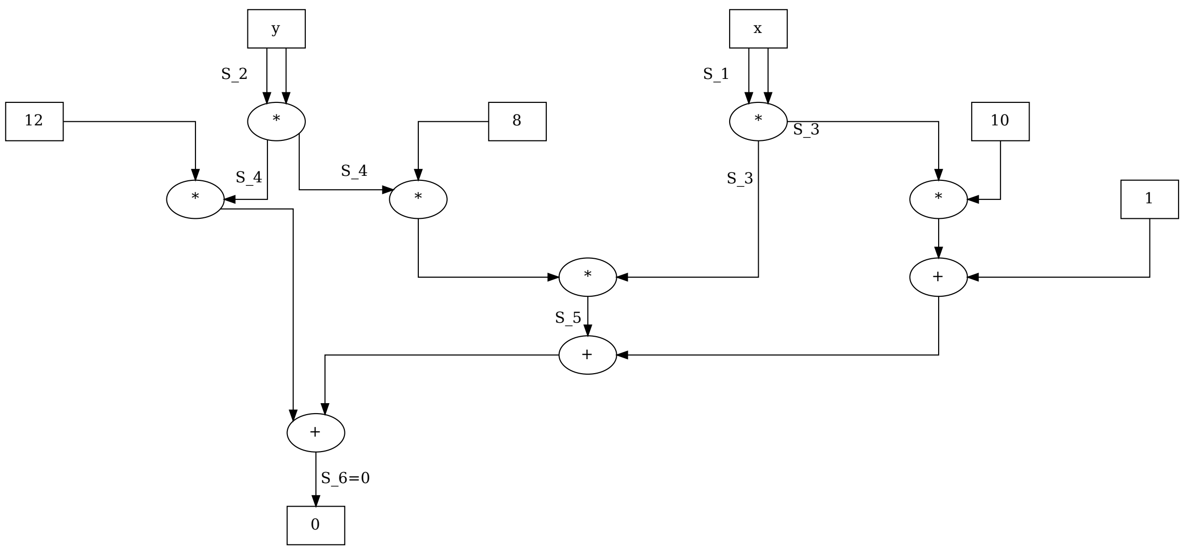
In the process of designing of circuits from functions, it should be noted that circuit representations are not unique in general. In case of the function $f_{tiny-jj}$, the circuit shape is dependent on our choice of bracketing in XXX. An alernative design i,s for example, given by the following circuit, which occurs when the bracketed expression $8\cdot((x\cdot x)\cdot(y\cdot y))$ is replaced by the expression $(x\cdot x)\cdot(8\cdot(y\cdot y))$.

add reference

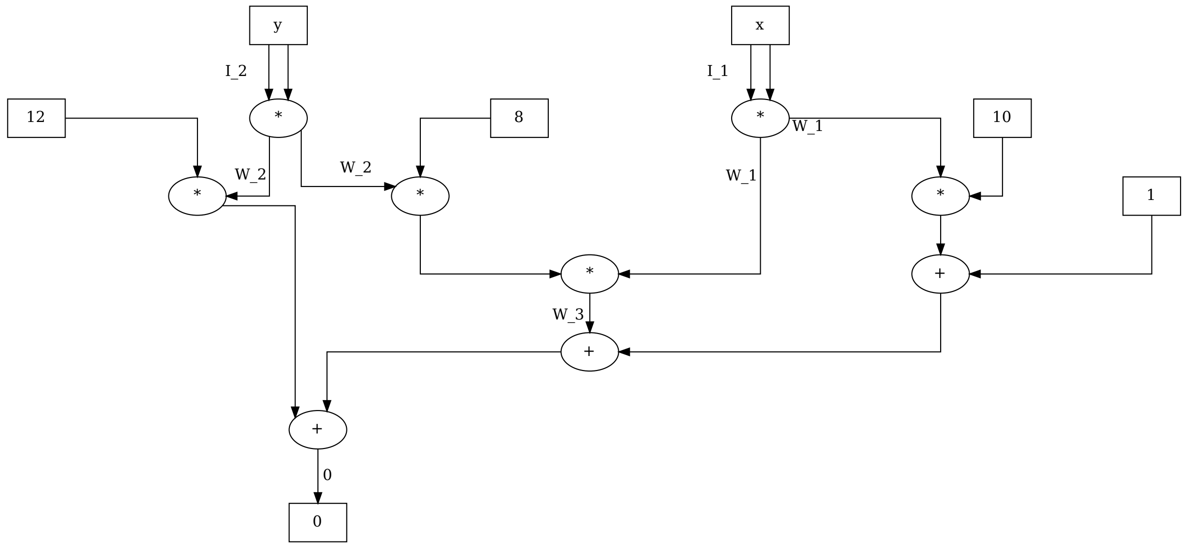


Of course, both circuits represent the same function, due to the associativity and commutativity laws that hold true in any field.

With a circuit that represents the function $f_{tiny-jj}$, we can now proceed to derive a circuit that constrains arbitrary pairs (x,y) of field elements to be points on the tiny-jubjub curve. To do so, we have to constrain the output to be zero, that is, we have to constrain $S_6 = 0$. To indicate this in the circuit, we replace the output variable by the constant 0 and constrain the related edge label accordingly. We get the following:



The previous circuit enforces input values assigned to the labels S_1 and S_2 to be points on the tiny-jubjub curve. However, it does not specify which labels are considered public and which are considered private. The following circuit defines the inputs to be public, while all other labels are private:



It can be shown that every space- and time-bounded computation can be transformed into an algebraic circuit. We call any process that transforms a bounded computation into a circuit **flattening**.

Circuit Execution Algebraic circuits are directed, acyclic multi-graphs, where nodes represent variables, constants, or addition and multiplication gates. In particular, every algebraic circuit with n input nodes decorated with variable symbols and m output nodes decorated with variables can be seen a function that transforms an input tuple $(x_1,\ldots,x_n)$ from $\mathbb{F}^n$ into an output tuple $(f_1,\ldots,f_m)$ from $\mathbb{F}^m$. The transformation is done by sending values associated to nodes along their outgoing edges to other nodes. If those nodes are gates, then the values are transformed according to the gate label and the process is repeated along all edges until a sink node is reached. We call this computation **circuit execution**.

When executing a circuit, it is possible to not only compute the output values of the circuit but to derive field elements for all edges, and, in particular, for all edge labels in the circuit. The result is a tuple $(S_1,S_2,\ldots,S_n)$ of field elements associated to all labeled edges, which we call a **valid assignment** to the circuit. In contrast, any assignment $(S'_1,S'_2,\ldots,S'_n)$ of field elements to edge labels that can not arise from circuit execution is called an **invalid assignment**.

Valid assignments can be interpreted as **proofs for proper circuit execution** because they keep a record of the computational result as well as intermediate computational steps.

Example 117 (3-factorization). Consider the 3-factorization problem from example 106 and its representation as an algebraic circuit from XXX. We know that the set of edge labels is given by $S := \{I_1; W_1, W_2, W_3, W_4\}$.

check reference

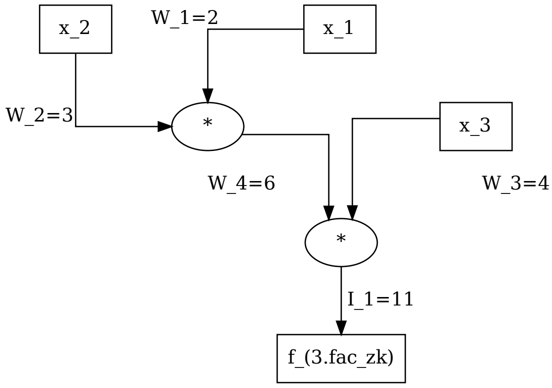
add reference

To understand how this circuit is executed, consider the variables $x_1 = 2$, $x_2 = 3$ as well as $x_3 = 4$. Following all edges in the graph, we get the assignments $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$. Then the assignments of W_1 and W_2 enter a multiplication gate and the output of the gate is $2\cdot 3 = 6$, which we assign to W_4, i.e. $W_4 = 6$. The values W_4 and W_3 then enter the second multiplication gate and the output of the gate is $6\cdot 4 = 11$, which we assign to I_1, i.e. $I_1 = 11$.

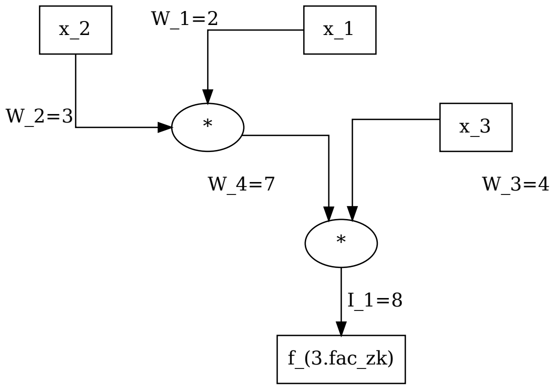
A valid assignment to the 3-factorization circuit $C_{3.fac}(\mathbb{F}_{13})$ is therefore given by the following set:

$$S_{valid} := \{11; 2, 3, 4, 6\} \tag{6.8}$$

We can visualise this assignment in the circuit as follows:



To see what an invalid assignment looks like, consider the assignment $S_{err} := \{8; 2, 3, 4, 7\}$. In this assignment, the input values are the same as in the previous case. The associated circuit is:



This assignment is invalid, as the assignments of I_1 and W_4 cannot be obtained by executing the circuit.

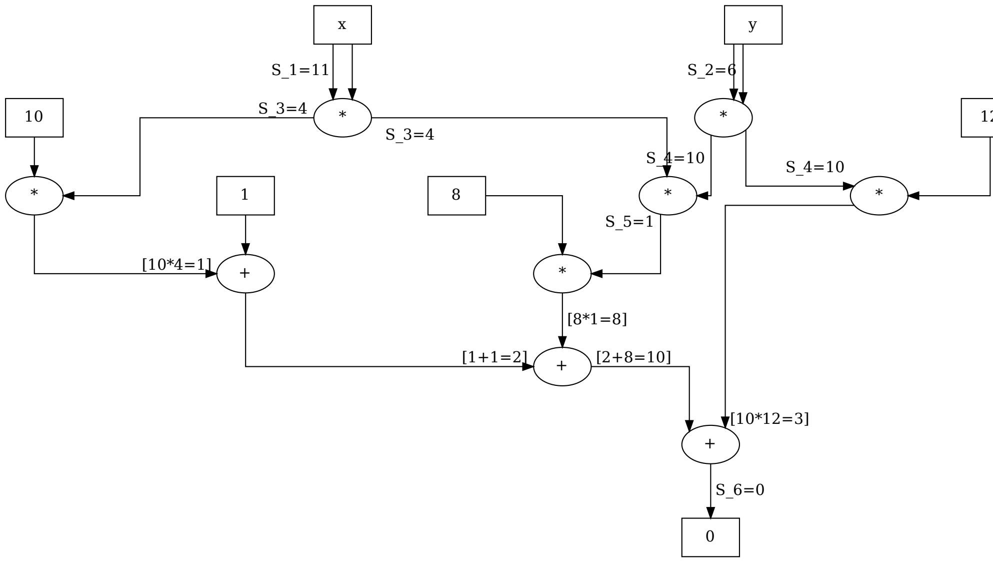
Example 118. To compute a more realistic algebraic circuit execution, consider the defining circuit $C_{tiny-jj}(\mathbb{F}_{13})$ from example 114 again. We already know from the way this circuit is constructed that any valid assignment with $S_1 = x$, $S_2 = y$ and $S_6 = 0$ will ensure that the pair (x, y) is a point on the tiny-jubjub curve in its Edwards representation (equation 5.20.

From example 114, we know that the pair $(11, 6)$ is a proper point on the tiny-jubjub curve and we use this point as input to a circuit execution. We get the following:

check reference

check reference

check reference



Executing the circuit, we indeed compute $S_6 = 0$ as expected, which proves that $(11,6)$ is a point on the tiny-jubjub curve in its Edwards representation. A valid assignment of $C_{tiny-jj}(\mathbb{F}_{13})$ is therefore given by the following equation:

$$S_{tiny-jj} = \{S_1, S_2, S_3, S_4, S_5, S_6\} = \{11, 6, 4, 10, 1, 0\}$$

Circuit Satisfiability To understand how algebraic circuits give rise to formal languages, observe that every algebraic circuit $C(\mathbb{F})$ over a fields $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{C(\mathbb{F})} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\} \; ; \; (I;W) \mapsto \begin{cases} true & (I;W) \text{is valid assignment to } C(\mathbb{F}) \\ false & else \end{cases} \tag{6.9}$$

Every algebraic circuit therefore defines a formal language. The grammar of this language is encoded in the shape of the circuit, words are assignments to edge labels that are derived from circuit execution, and **statements** are knowledge claims "Given instance I, there is a witness W such that $(I;W)$ is a valid assignment to the circuit". A constructive proof to this claim is therefore an assignment of a field element to every witness variable, which is verified by executing the circuit to see if the assignment of the execution meets the assignment of the proof.

In the context of zero-knowledge proof systems, executing circuits is also often called **witness generation**, since in applications the instance part is usually public, while its the task of a proofer to compute the witness part.

Remark 4 (Circuit satisfiability). It should be noted that, in our definition, every circuit defines its own language. However, in more theoretical approaches another language usually called **circuit satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness, or computational complexity of the class of **all** algebraic circuits over a given field. From our perspective the circuit satisfiability language is obtained by union of all circuit languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

Should we refer to R1CS satisfiability (p. 140 here?

$$L_{CIRCUIT\_SAT(\mathbb{F})} = \{(i;w) \in \Sigma^* \times \Sigma^* \mid \text{there is a circuit } C(\mathbb{F}) \text{ such that } (i;w) \text{ is valid assignment}\}$$

Example 119 (3-Factorization). Consider the circuit $C_{3.fac}$ from equation 6.8 again. We call the associated language $L_{3.fac\_circ}$. check reference

To understand how a constructive proof of a statement in $L_{3.fac\_circ}$ looks like, consider the instance $I_1 = 11$. To provide a proof for the statement "There exist a witness W such that $(I_1;W)$ is a word in $L_{3.fac\_circ}$" a proof therefore has to consists of proper values for the variables W_1, W_2, W_3 and W_4. Any proofer therefore has to find input values for W_1, W_2 and W_3 and then execute the circuit to compute W_4 under the assumption $I_1 = 11$.

Example XXX implies that $(2,3,4,6)$ is a proper constructive proof and in order to verify the proof a verifier needs to execute the circuit with instance $I_1 = 11$ and inputs $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$ to decide whether the proof is a valid assignment or not. add reference

Associated Constraint Systems As we have seen in 6.2.1.1, rank-1 constraint systems define a way to represent statements in terms of a system of quadratic equations over finite fields, suitable for pairing-based zero-knowledge proof systems. However, those equations provide no practical way for a proofer to actually compute a solution. On the other hand, algebraic circuits can be executed in order to derive valid assignments efficiently. check reference

In this paragraph, we show how to transform any algebraic circuit into a rank-1 constraint system such that valid circuit assignments are in 1:1 correspondence with solutions to the associated R1CS.

To see this, let $C(\mathbb{F})$ be an algebraic circuit over a finite field $\mathbb{F}$, with a set of edge labels $\{S_1, S_2, \ldots, S_n\}$. Then one of the following steps is executed for every edge label S_j from that set:

Definition 6.2.2.2.

- If the edge label S_j is an outgoing edge of a multiplication gate, the R1CS gets a new quadratic constraint

$$(\text{left input}) \cdot (\text{right input}) = S_j \tag{6.10}$$

  where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate, and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

- If the edge label S_j is an outgoing edge of an addition gate, the R1CS gets a new quadratic constraint

$$(\text{left input} + \text{right input}) \cdot 1 = S_j \tag{6.11}$$

  where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

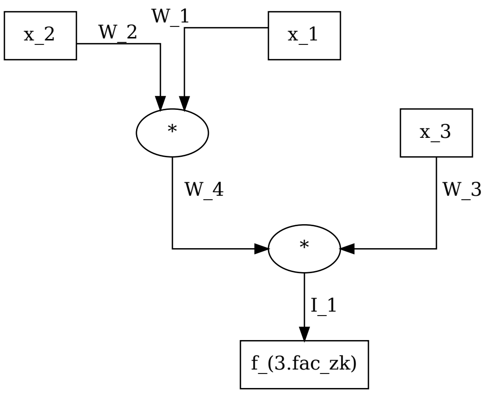
- No other edge label adds a constraint to the system.

The result of this method is a rank-1 constraint system, and, in this sense, every algebraic circuit $C(\mathbb{F})$ generates a R1CS R, which we call the **associated R1CS** of the circuit. It can be shown that a tuple of field elements $(S_1, S_2, \ldots, S_n)$ is a valid assignment to a circuit if and only if the same tuple is a solution to the associated R1CS. Circuit executions therefore compute solutions to rank-1 constraints systems efficiently.

To understand the contribution of algebraic gates to the number of constraints, note that, by definition, multiplication gates have labels on their outgoing edges if and only if there is at least one labeled edge in both input paths, or if the outgoing edge is an input to a sink node. This implies that multiplication with a constant is essentially free in the sense that it doesn't add a new constraint to the system, as long as that multiplication gate is not am input to an output node.

Moreover, addition gates have labels on their outgoing edges if and only if they are inputs to sink nodes. This implies that addition is essentially free in the sense that it doesn't add a new constraint to the system, as long as that addition gate is not an input to an output node.

Example 120 (3-factorization). Consider our 3-factorization problem from equation 6.8 and the associated circuit $C_{3.fac}(\mathbb{F}_{13})$. Our task is to transform this circuit into an equivalent rank-1 constraint system. check reference



We start with an empty R1CS, and, in order to generate all constraints, we have to iterate over the set of edge labels $\{I_1; W_1, W_2, W_3, W_4\}$.

Starting with the edge label I_1, we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$\begin{aligned}(\text{left input}) \cdot (\text{right input}) &= I_1 && \Leftrightarrow\\ W_4 \cdot W_3 &= I_1\end{aligned}$$

Next, we consider the edge label W_1 and, since, it's not an outgoing edge of a multiplication or addition label, we don't add a constraint to the system. The same holds true for the labels W_2 and W_3.

For edge label W_4 , we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

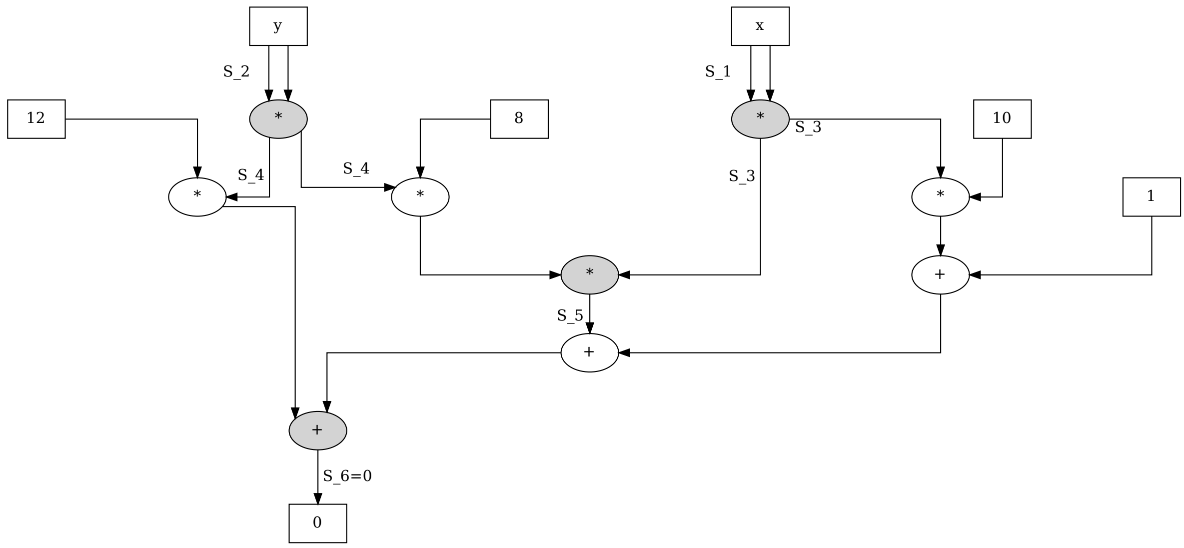
$$\begin{aligned}(\text{left input}) \cdot (\text{right input}) &= W_4 && \Leftrightarrow\\ W_2 \cdot W_1 &= W_4\end{aligned}$$

Since there are no more labeled edges, all constraints are generated, and we have to combine them to get the associated R1CS of $C_{3.fac}(\mathbb{F}_{13})$:

$$\begin{aligned}W_4 \cdot W_3 &= I_1\\ W_2 \cdot W_1 &= W_4\end{aligned}$$

This system is equivalent to the R1CS we derived in example 111. The languages $L_{3.fac\_zk}$ and $L_{3.fac\_circ}$ are therefore equivalent and both the circuit as well as the R1CS are just two different ways of expressing the same language. check reference

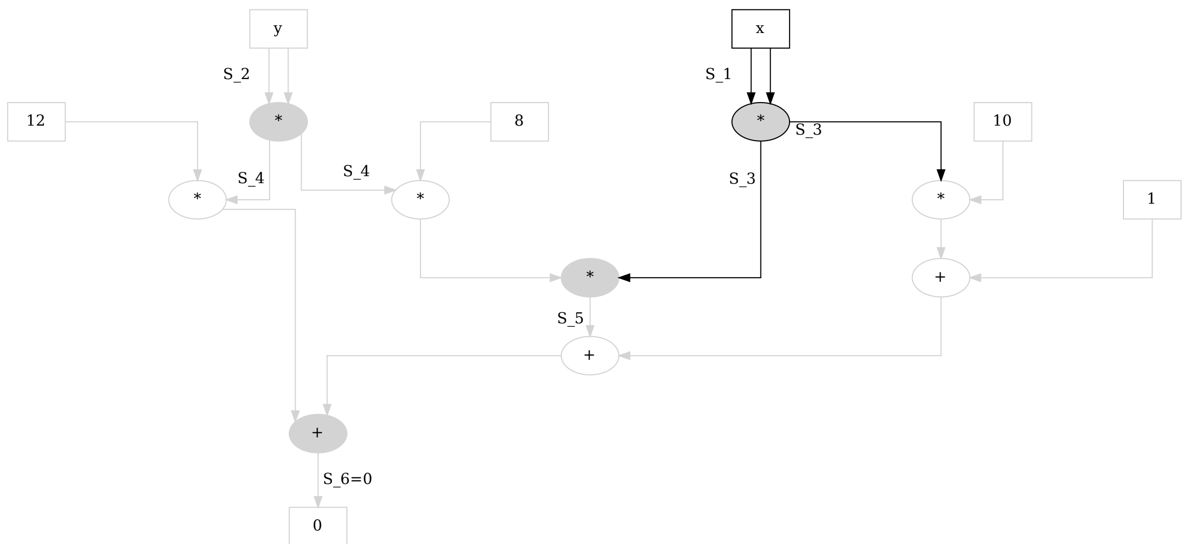
Example 121. To consider a more general transformation, we consider the tiny-jubjub circuit from example 114 again. A proper circuit is given by check reference



To compute the number of constraints, observe that we have 3 multiplication gates that have labels on their outgoing edges and 1 addition gate that has a label on its outgoing edge. We therefore have to compute 4 quadratic constraints.

In order to derive the associated R1CS, we have start with an empty R1CS and then iterate over the set $\{S_1, S_2, S_3, S_4, S_5, S_6 = 0\}$ of all edge labels, in order to generate the constraints.

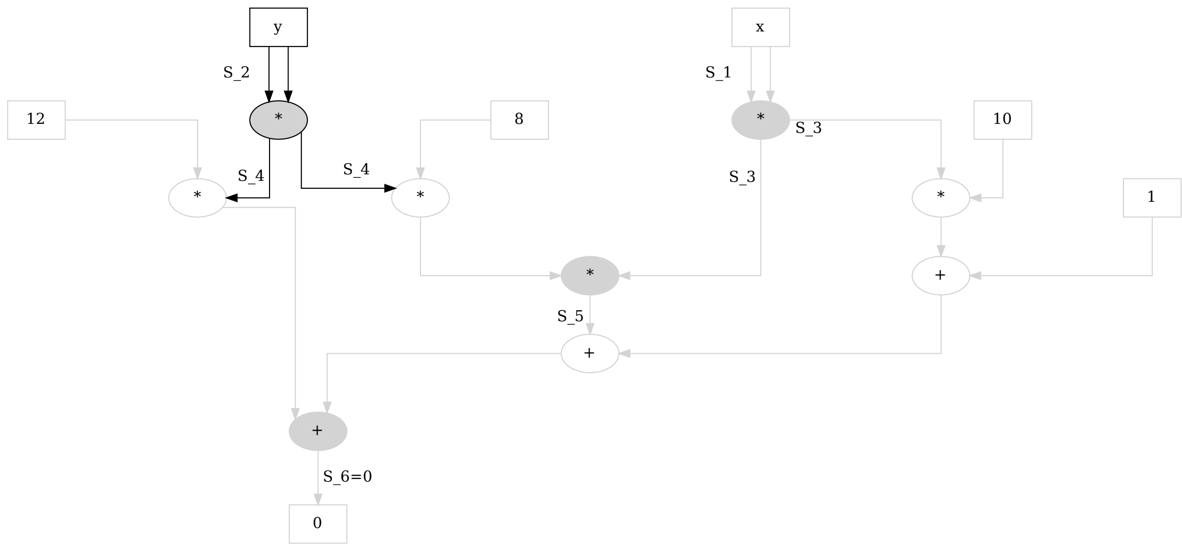
Considering edge label S_1, we see that the associated edges are not outgoing edges of any algebraic gate, and we therefore have to add no new constraint to the system. The same holds true for edge label S_2. Looking at edge label S_3, we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:



Both the left and the right input to this multiplication gate are labeled by S_1. We therefore have to add the following constraint to the system:

$$S_1 \cdot S_1 = S_3$$

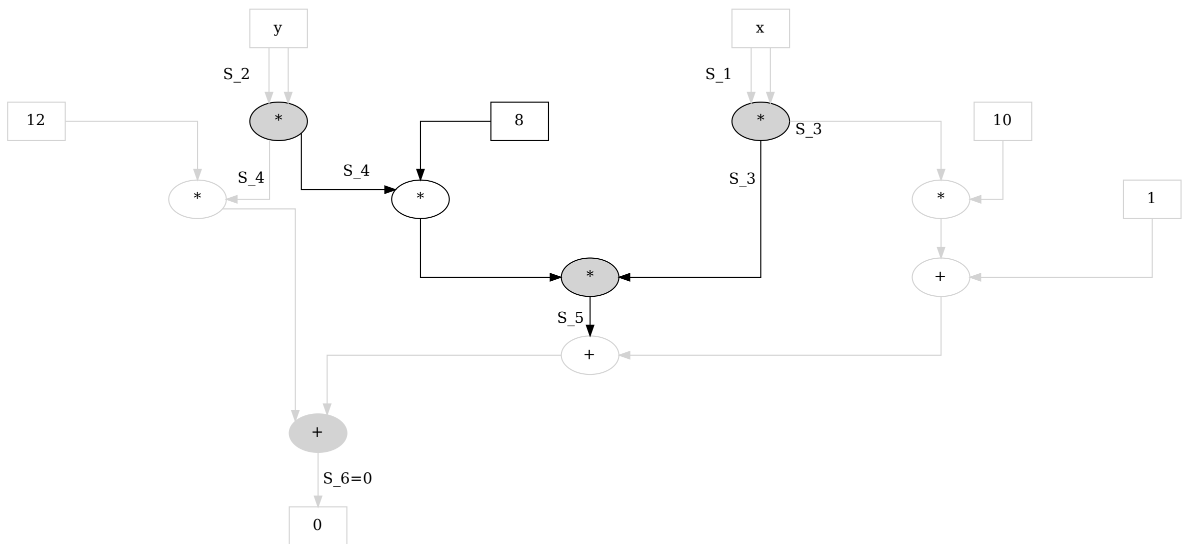
Looking at edge label S_4, we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:



Both the left and the right input to this multiplication gate are labeled by S_2 and we therefore have to add the following constraint to the system:

$$S_2 \cdot S_2 = S_4$$

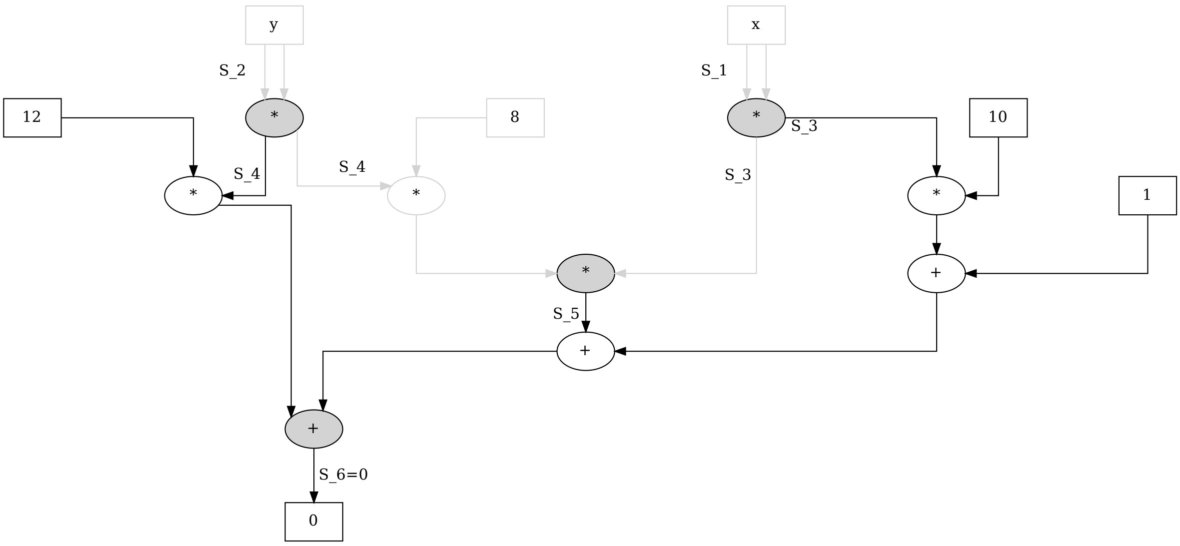
Edge label S_5 is more interesting. To see if it implies a constraint, we have to construct the associated subgraph first, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get



The right input to the associated multiplication gate is given by the labeled edge S_3. However, the left input is not a labeled edge, but has a labeled edge in one of its path. This implies that we have to add a constraint to the system. To compute the left factor of that constraint, we have to compute the output of subgraph associated to the left edge, which is $8 \cdot W_2$. This gives the constraint

$$(S_4 \cdot 8) \cdot S_3 = S_5$$

The last edge label is the constant $S_6 = 0$. To see if it implies a constraint, we have to construct the associated subgraph, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get



Both the left and the right input are unlabeled, but have a labeled edges in their path. This implies that we have to add a constraint to the system. Since the gate is an addition gate, the right factor in the quadratic constraint is always 1 and the left factor is computed by symbolically executing all inputs to all gates in sub-circuit. We get

$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

Since there are no more labeled outgoing edges, we are done deriving the constraints. Combining all constraints together, we get the following R1CS:

$$\begin{aligned} S_1 \cdot S_1 &= S_3 \\ S_2 \cdot S_2 &= S_4 \\ (S_4 \cdot 8) \cdot S_3 &= S_5 \\ (12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 &= 0 \end{aligned}$$

which is equivalent to the R1CS we derived in example 114. The languages $L_{3.fac\_zk}$ and $L_{3.fac\_circ}$ are therefore equivalent and both the circuit as well as the R1CS are just two different ways to express the same language.

check reference

6.2.3 Quadratic Arithmetic Programs

We have introduced algebraic circuits and their associated rank-1 constraints systems as two particular models able to represent space- and time-bounded computation. Both models define formal languages, and associated membership as well as knowledge claims can be constructively proofed by executing the circuit in order to compute solutions to its associated R1CS.

One reason why those systems are useful in the context of succinct zero-knowledge proof systems is because any R1CS can be transformed into another computational model called **quadratic arithmetic programs** (QAP), which serve as the basis for some of the most efficient succinct non-interactive zero-knowledge proof generators that currently exist.

As we will see, proving statements for languages that have checking relations defined by quadratic arithmetic programs can be achieved by providing certain polynomials, and those proofs can be verified by checking a particular divisibility property.

QAP representation To understand what quadratic arithmetic programs are in detail, let $\mathbb{F}$ be a field and R a rank-1 constraints system over $\mathbb{F}$ such that the number of non-zero elements in $\mathbb{F}$ is strictly larger then the number k of constraints in R. Moreover, let a^i_j, b^i_j and $c^i_j \in \mathbb{F}$ for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$, be the defining constants of the R1CS and $m_1, \ldots, m_k$ be arbitrary, invertible and distinct elements from $\mathbb{F}$.

Then a **quadratic arithmetic program** [QAP] of the R1CS is the following set of polynomials over $\mathbb{F}$:

$$QAP(R) = \left\{ T \in \mathbb{F}[x], \left\{ A_j, B_j, C_j \in \mathbb{F}[x] \right\}_{h=0}^{n+m} \right\} \tag{6.12}$$

In the equation above, $T(x) := \Pi_{l=1}^{k}(x - m_l)$ is a polynomial of degree k, called the **target polynomial** of the QAP and A_j, B_j as well as C_j are the unique degree $k-1$ polynomials defined by the following equation:

$$A_j(m_i) = a^i_j \quad B_j(m_i) = b^i_j \quad C_j(m_i) = C^i_j \quad j = 1, \ldots, n+m+1, i = 1, \ldots, k \tag{6.13}$$

Given some rank-1 constraint system, an associated quadratic arithmetic program is therefore nothing but a set of polynomials, computed from the constants in the R1CS. To see that the polynomials A_j, B_j and C_j are uniquely defined by the equations in XXX, recall that a a polynomial of degree $k-1$ is completely determined on k evaluation points and the equation 4 precisely determines those k evaluation points.

add reference

"by"?

Since we only consider polynomials over fields, Lagrange's interpolation method from 3.30 in chapter 3 can be used to derive the polynomials A_j, B_j and C_j from their defining equations XXX. A practical method to compute a QAP from a given R1CS therefore consists of two steps. If the R1CS consists of k constraints, first choose k invertible and mutually different points from the underlaying field. Every choice defines a different QAP for the same R1CS. Then use Lagrange's method and equation XXX to compute the polynomials A_j, B_j and C_j for every $1 \leq j \leq k$.

check reference

check reference

add reference

Example 122 (Generalized factorization SNARK). To provide a better intuition of quadratic arithmetic programs and how they are computed from their associated rank-1 constraint systems, consider the language $L_{3.fac\_zk}$ from example 106 and its associated R1CS:

add reference

check reference

$$\begin{aligned} W_1 \cdot W_2 &= W_4 && \text{constraint 1} \\ W_4 \cdot W_3 &= I_1 && \text{constraint 2} \end{aligned}$$

In this example we want to transform this R1CS into an associated QAP. In a first step, we have to compute the defining constants a^i_j, b^i_j and c^i_j of the R1CS. According to XXX, we have

add reference

$$
\begin{array}{llllll}
a_0^1=0 & a_1^1=0 & a_2^1=1 & a_3^1=0 & a_4^1=0 & a_5^1=0 \\
a_0^2=0 & a_1^2=0 & a_2^2=0 & a_3^2=0 & a_4^2=0 & a_5^2=1 \\
\\
b_0^1=0 & b_1^1=0 & b_2^1=0 & b_3^1=1 & b_4^1=0 & b_5^1=0 \\
b_0^2=0 & b_1^2=0 & b_2^2=0 & b_3^2=0 & b_4^2=1 & b_5^2=0 \\
\\
c_0^1=0 & c_1^1=0 & c_2^1=0 & c_3^1=0 & c_4^1=0 & c_5^1=1 \\
c_0^2=0 & c_1^2=1 & c_2^2=0 & c_3^2=0 & c_4^2=0 & c_5^2=0
\end{array}
$$

Since the R1CS is defined over the field $\mathbb{F}_{13}$ and has two constraining equations, we need to choose two arbitrary but distinct elements m_1 and m_2 from $\mathbb{F}_{13}$. We choose $m_1 = 5$, and $m_2 = 7$ and with this choice we get the target polynomial

$$
\begin{aligned}
T(x) &= (x-m_1)(x-m_2) && \text{\# Definition of T} \\
&= (x-5)(x-7) && \text{\# Insert our choice} \\
&= (x+8)(x+6) && \text{\# Negatives in } \mathbb{F}_{13} \\
&= x^2+x+9 && \text{\# expand}
\end{aligned}
$$

Then we have to compute the polynomials A_j, B_j and C_j by their defining equation from the R1CS coefficients. Since the R1CS has two constraining equations, those polynomials are of degree 1 and they are defined by their evaluation at the point $m_1 = 5$ and the point $m_2 = 7$.

At point m_1, each polynomial A_j is defined to be a_j^1 and at point m_2, each polynomial A_j is defined to be a_j^2. The same holds true for the polynomials B_j as well as C_j. Writing all these equations now, we get:

$$
\begin{array}{llllll}
A_0(5)=0, & A_1(5)=0, & A_2(5)=1, & A_3(5)=0, & A_4(5)=0, & A_5(5)=0 \\
A_0(7)=0, & A_1(7)=0, & A_2(7)=0, & A_3(7)=0, & A_4(7)=0, & A_5(7)=1 \\
\\
B_0(5)=0, & B_1(5)=0, & B_2(5)=0, & B_3(5)=1, & B_4(5)=0, & B_5(5)=0 \\
B_0(7)=0, & B_1(7)=0, & B_2(7)=0, & B_3(7)=0, & B_4(7)=1, & B_5(7)=0 \\
\\
C_0(5)=0, & C_1(5)=0, & C_2(5)=0, & C_3(5)=0, & C_4(5)=0, & C_5(5)=1 \\
C_0(7)=0, & C_1(7)=1, & C_2(7)=0, & C_3(7)=0, & C_4(7)=0, & C_5(7)=0
\end{array}
$$

Lagrange's interpolation implies that a polynomial of degree k, that is, that zero on $k+1$ points has to be the zero polynomial. Since our polynomials are of degree 1 and determined on 2 points, we therefore know that the only non-zero polynomials in our QAP are A_2, A_5, B_3, B_4, C_1 and C_5, and that we can use Lagrange's interpolation to compute them.

To compute A_2 we note that the set S in our version of Lagrange's method is given by $S = \{(x_0,y_0),(x_1,y_1)\} = \{(5,1),(7,0)\}$. Using this set we get:

$$
\begin{aligned}
A_2(x) &= y_0\cdot l_0 + y_1\cdot l_1 \\
&= y_0\cdot\left(\frac{x-x_1}{x_0-x_1}\right) + y_1\cdot\left(\frac{x-x_0}{x_1-x_0}\right) = 1\cdot\left(\frac{x-7}{5-7}\right)+0\cdot\left(\frac{x-5}{7-5}\right) \\
&= \frac{x-7}{-2} = \frac{x-7}{11} && \text{\# } 11^{-1}=6 \\
&= 6(x-7) = 6x+10 && \text{\# } -7=6 \text{ and } 6\cdot 6=10
\end{aligned}
$$

To compute A_5, we note that the set S in our version of Lagrange's method is given by $S = \{(x_0,y_0),(x_1,y_1)\} = \{(5,0),(7,1)\}$. Using this set we get:

$$\begin{aligned}
A_5(x) &= y_0 \cdot l_0 + y_1 \cdot l_1 \\
&= y_0 \cdot (\frac{x-x_1}{x_0-x_1}) + y_1 \cdot (\frac{x-x_0}{x_1-x_0}) = 0 \cdot (\frac{x-7}{5-7}) + 1 \cdot (\frac{x-5}{7-5}) \\
&= \frac{x-5}{2} && \# \, 2^{-1} = 7 \\
&= 7(x-5) = 7x+4 && \# \, -5 = 8 \text{ and } 7 \cdot 8 = 4
\end{aligned}$$

Using Lagrange's interpolation, we can deduce that $A_2 = B_3 = C_5$ as well as $A_5 = B_4 = C_1$, since they are polynomials of degree 1 that evaluate to same values on 2 points. Using this, we get the following set of polynomials

$A_0(x) = 0$	$B_0(x) = 0$	$C_0(x) = 0$
$A_1(x) = 0$	$B_1(x) = 0$	$C_1(x) = 7x+4$
$A_2(x) = 6x+10$	$B_2(x) = 0$	$C_2(x) = 0$
$A_3(x) = 0$	$B_3(x) = 6x+10$	$C_3(x) = 0$
$A_4(x) = 0$	$B_4(x) = 7x+4$	$C_4(x) = 0$
$A_5(x) = 7x+4$	$B_5(x) = 0$	$C_5(x) = 6x+10$

We can invoke Sage to verify our computation. In sage every polynomial ring has a function `lagrange_polynomial` that takes the defining points as inputs and the associated Lagrange polynomial as output.

```
sage: F13 = GF(13)
sage: F13t.<t> = F13[]
sage: T = F13t((t-5)*(t-7))
sage: A2 = F13t.lagrange_polynomial([(5,1),(7,0)])
sage: A5 = F13t.lagrange_polynomial([(5,0),(7,1)])
sage: T == F13t(t^2 + t + 9)
True
sage: A2 == F13t(6*t + 10)
True
sage: A5 == F13t(7*t + 4)
True
```

Combining this computation with the target polynomial we derived earlier, a quadratic arithmetic program associated to the rank-1 constraint system $R_{3.fac\_zk}$ is given as follows:

$$\begin{aligned}
QAP(R_{3.fac_zk}) = \{x^2+x+9, \\
\{0,0,6x+10,0,0,7x+4\},\{0,0,0,6x+10,7x+4,0\},\{0,7x+4,0,0,0,6x+10\}\}
\end{aligned} \tag{6.14}$$

QAP Satisfiability One of the major points of quadratic arithmetic programs in proofing systems is that solutions of their associated rank-1 constraints systems are in 1:1 correspondence with certain polynomials P such that P is divisible by the target polynomial T of the QAP if and only if the solution id a solution. Verifying solutions to the R1CS and hence, checking proper circuit execution is then achievable by polynomial division of P by T.

clarify language

To be more specific, let R be some rank-1 constraints system with associated assignment variables $(I_1,\ldots,I_n;W_1,\ldots,W_m)$ and let $QAP(R)$ be a quadratic arithmetic program of R. Then the tuple $(I_1,\dots,I_n;W_1,\dots,W_m)$ is a solution to the R1CS if and only if the following polynomial is divisible by the target polynomial T:

$$P_{(I;W)} = \left(A_0+\Sigma_j^n I_j\cdot A_j+\Sigma_j^m W_j\cdot A_{n+j}\right)\cdot\left(B_0+\Sigma_j^n I_j\cdot B_j+\Sigma_j^m W_j\cdot B_{n+j}\right)-\left(C_0+\Sigma_j^n I_j\cdot C_j+\Sigma_j^m W_j\cdot C_{n+j}\right) \tag{6.15}$$

Every tuple $(I;W)$ defines a polynomial $P_{(I;W)}$, and, since each polynomial A_j, B_j and C_j is of degree $k-1$, $P_{(I;W)}$ is of degree $(k-1)\cdot(k-1)=k^2-2k+1$.

To understand how quadratic arithmetic programs define formal languages, observe that every QAP over a field $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F}\times\mathbb{F}$ in the following way:

$$R_{QAP} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\}\ ;\ (I;W) \mapsto \begin{cases} true & P_{(I;W)} \text{ is divisible by } T \\ false & else \end{cases} \tag{6.16}$$

This means that every QAP defines a formal language, and, if the QAP is associated to an R1CS, it can be shown that the two languages are equivalent. A **statement** is a membership claim "There is a word $(I;W)$ in L_{QAP}". A proof to this claim is therefore a polynomial $P_{(I;W)}$, which is verified by dividing $P_{(I;W)}$ by T.

Note the structural similarity to the definition of an R1CS in 6.2.1.1 and the different ways of computing proofs in both systems. For circuits and their associated rank-1 constraints systems, a constructive proof consists of a valid assignment of field elements to the edges of the circuit, or the variables in the R1CS. However, in the case of QAPs, a valid proof consists of a polynomial $P_{(I;W)}$.

check reference

To compute a proof for a statement in L_{QAP} given some instance I, a proofer first needs to compute a constructive proof W, e.g. by executing the circuit. With $(I;W)$ at hand, the proofer can then compute the polynomial $P_{(I;W)}$ and publish it as proof.

Verifying a constructive proof in the case of a circuit is achieved by executing the circuit, comparing the result to the given proof, and verifying the same proof in the R1CS picture means checking if the elements of the proof satisfy all equation.

In contrast, verifying a proof in the case of a QAP is done by polynomial division of the proof P by the target polynomial T of the QAP. The proof checks out if and only if P is divisible by T.

Example 123. Consider the quadratic arithmetic program $QAP(R_{3.fac\_zk})$ from example XXX and its associated R1CS from equation 6.1.0.1. To give an intuition of how proofs in the language $L_{QAP(R_{3.fac\_zk})}$ lets consider the instance $I_1 = 11$. As we know from example XXX, $(W_1,W_2,W_3,W_5)=(2,3,4,6)$ is a proper witness, since $(I_1;W_1,W_2,W_3,W_5)=(11;2,3,4,6)$ is a valid circuit assignment and hence, a solution to $R_{3.fac\_zk}$ and a constructive proof for language $L_{R_{3.fac\_zk}}$.

add reference

check reference

add reference

In order to transform this constructive proof into a membership proof in language $L_{QAP(R_{3.fac\_zk})}$, a proofer has to use the elements of the constructive proof, to compute the polynomial $P_{(I;W)}$.

In the case of $(I_1;W_1,W_2,W_3,W_5)=(11;2,3,4,6)$, the associated proof is computed as follows:

$$
\begin{aligned}
P_{(I;W)} =& \left(A_0+\sum_j^n I_j\cdot A_j+\sum_j^m W_j\cdot A_{n+j}\right)\cdot\left(B_0+\sum_j^n I_j\cdot B_j+\sum_j^m W_j\cdot B_{n+j}\right)-\left(C_0+\sum_j^n I_j\cdot C_j+\sum_j^m W_j\cdot C_{n+j}\right)\\
=&(2(6x+10)+6(7x+4))\cdot(3(6x+10)+4(7x+4))-(11(7x+4)+6(6x+10))\\
=&((12x+7)+(3x+11))\cdot((5x+4)+(2x+3))-((12x+5)+(10x+8))\\
=&(2x+5)\cdot(7x+7)-(9x)\\
=&(x^2+2\cdot 7x+5\cdot 7x+5\cdot 7)-(9x)\\
=&(x^2+x+9x+9)-(9x)\\
=&x^2+x+9
\end{aligned}
$$

Given instance $I_1 = 11$ a proofer therefore provides the polynomial x^2+x+9 as proof. To verify this proof, any verifier can then look up the target polynomial T from the QAP and divide $P_{(I;W)}$ by T. In this particular example, $P_{(I;W)}$ is equal to the target polynomial T, and hence, it is divisible by T with $P/T = 1$. The verification therefore checks the proof.

```
sage: F13 = GF(13)
sage: F13t.<t> = F13[]
sage: T = F13t(t^2 + t + 9)
sage: P = F13t((2*(6*t+10)+6*(7*t+4))*(3*(6*t+10)+4*(7*t +4))
  -(11*(7*t+4)+6*(6*t+10)))
sage: P == T
True
sage: P % T # remainder
0
```

To give an example of a false proof, consider the tuple $(I_1;W_1,W_2,W_3,W_4) = (11,2,3,4,8)$. Executing the circuit, we can see that this is not a valid assignment and not a solution to the R1CS, and hence, not a constructive knowledge proof in $L_{3.fac\_zk}$. However, a proofer might use these values to construct a false proof $P_{(I;W)}$:

$$
\begin{aligned}
P'_{(I;W)} =& \left(A_0+\sum_j^n I_j\cdot A_j+\sum_j^m W_j\cdot A_{n+j}\right)\cdot\left(B_0+\sum_j^n I_j\cdot B_j+\sum_j^m W_j\cdot B_{n+j}\right)-\left(C_0+\sum_j^n I_j\cdot C_j+\sum_j^m W_j\cdot C_{n+j}\right)\\
=&(2(6x+10)+8(7x+4))\cdot(3(6x+10)+4(7x+4))-(8(6x+10)+11(7x+4))\\
=&8x^2+6
\end{aligned}
$$

Given instance $I_1 = 11$, a proofer therefore provides the polynomial $8x^2+6$ as proof. To verify this proof, any verifier can look up the target polynomial T from the QAP and divide $P_{(I;W)}$ by T. However, polynomial division has the following remainder:

$$(8x^2+6)/(x^2+x+9) = 8+\frac{5x+12}{x^2+x+9}$$

This implies that $P_{(I;W)}$ is not divisible by T, and hence, the verification does not check the proof. Any verifier can therefore show that the proof is false.

```
sage: F13 = GF(13)
sage: F13t.<t> = F13[]
sage: T = F13t(t^2 + t + 9)
sage: P = F13t((2*(6*t+10)+8*(7*t+4))*(3*(6*t+10)+4*(7*t+4))-(
  8*(6*t+10)+11*(7*t+4)))
```

```
sage: P == F13t(8*t^2 + 6)
True
sage: P % T # remainder
5*t + 12
```

Chapter 7

Circuit Compilers

As we have seen in the previous chapter, statements can be formalized as membership or knowledge claims in formal language, and algebraic circuits as well as rank-1 constraint systems are two practically important ways to define those languages.

However, both algebraic circuits and rank-1 constraint systems are not ideal from a developers point of view, because they deviate substantially from common programing paradigms. Writing real-world applications as circuits and the associated verification in terms of rank-1 constraint systems is as least as troublesome as writing any other low-level language like assembler code. To allow for complex statement design, it is therefore necessary to have some kind of compiler framework, capable of transforming high-level languages into arithmetic circuits and associated rank-1 constraint systems.

As we have seen in chapter 6 and in 6.2.1.1,, both arithmetic circuits and rank-1 constraint systems have a modularity property by which it is possible to synthesize complex circuits from simple ones. A basic approach taken by many circuit/R1CS compilers is therefore to provide a library of atomic and simple circuits and then define a way to combine those basic building blocks into arbitrary complex systems.

check references

In this chapter, we provide an introduction to basic concepts of so-called **circuit compilers** and derive a toy language which we can "compile" in a pen-and-paper approach into algebraic circuits and their associated rank-1 constraint systems.

We start with a general introduction to our language, and then introduce atomic types like booleans and unsigned integers. Then we define the fundamental control flow primitives like the if-then-else conditional and the bounded loop. We will look at basic functionality primitives like elliptic curve cryptography. Primitives like these are often called **gadgets** in the literature.

7.1 A Pen-and-Paper Language

To explain basic concepts of circuit compilers and their associated high-level languages, we derive an informal toy language and associated "brain-compiler" which we name `PAPER` (**P**en-**A**nd-**P**aper **E**xecution **R**ules). `PAPER` allows programmers to define statements in Rust-like pseudo-code. The language is inspired by `ZOKRATES` and `circom`.

add references to these languages?

7.1.1 The Grammar

In `PAPER`, any statement is defined as an ordered list of functions, where any function has to be declared in the list before it is called in another function of that list. The last entry in a statement has to be a special function, called `main`. Functions take a list of typed parameters as inputs and compute a tuple of typed variables as output, where types are special functions that define how to transform that type into another type, ultimately transforming any type into elements of the base field where the circuit is defined over.

Any statement is parameterized over the field that the circuit will be defined on, and has additional optional parameters of unsigned type, needed to define the size of array or the counter of bounded loops. The following definition makes the grammar of a statement precise using a command line language like description:

```
statement <Name> {F:<Field> [ , <N_1: unsigned>,... ] } {
  [fn <Name>([[pub]<Arg>:<Type>,...]) -> (<Type>,...){
    [let [pub] <Var>:<Type> ;... ]
    [let const <Const>:<Type>=<Value> ;... ]
    Var<==(fn([<Arg>|<Const>|<Var>,...])|(<Arg>|<Const>|<Var>)) ;
    return (<Var>,...) ;
  } ;...]
  fn main([[pub]<Arg>:<Type>,...]) -> (<Type>,...){
    [let [pub] <Var>:<Type> ;... ]
    [let const <Const>:<Type>=<Value> ;... ]
    Var<==(fn([<Arg>|<Const>|<Var>,...])|(<Arg>|<Const>|<Var>)) ;
    return (<Var>,...) ;
  } ;
}
```

Function arguments and variables are private by default, but can be declared as public by the `pub` specifier. Declaring arguments and variables as public always overwrites any previous or conflicting private declarations. Every argument, constant or variable has a type, and every type is defined as a function that transforms that type into another type:

```
type <TYPE>( t1 : <TYPE_1>) -> TYPE_2{
  let t2: TYPE_2 <== fn(TYPE_1)
  return t2
}
```

Many real-world circuit languages are based on a similar, but of course more sophisticated approach than `PAPER`. The purpose of `PAPER` is to show basic principles of circuit compilers and their associated high-level languages.

Example 124. To get a better understanding of the grammar of `PAPER`, the following constitutes proper high-level code that follows the grammar of the `PAPER` language, assuming that all types in that code have been defined elsewhere.

```
statement MOCK_CODE {F: F_43, N_1 = 1024, N_2 = 8} {
  fn foo(in_1 : F, pub in_2 : TYPE_2) -> F {
    let const c_1 : F = 0 ;
    let const c_2 : TYPE_2 = SOME_VALUE ;
    let pub out_1 : F ;
    out_1<== c_1 ;
    return out_1 ;
  } ;

  fn bar(pub in_1 : F) -> F {
    let out_1 : F ;
    out_1<==foo(in_1);
    return out_1 ;
```

```
  } ;

  fn main(in_1 : TYPE_1)->(F, TYPE_2){
    let const c_1 : TYPE_1  = SOME_VALUE ;
    let const c_2 : F = 2;
    let const c_3 : TYPE_2  = SOME_VALUE  ;
    let pub out_1 : F ;
    let out_2 : TYPE_2 ;
    c_1 <== in_1 ;
    out_1 <== foo(c_2) ;
    out_2 <== TYPE_2 ;
    return (out_1,out_2) ;
  } ;
}
```

7.1.2 The Execution Phases

In contrast to normal executable programs, programs for circuit compilers have two modes of execution. The first mode, usually called the **setup phase**, is executed in order to generate the circuit and its associated rank-1 constraint system, the latter of which is then usually used as input to some zero-knowledge proof system.

The second mode of execution is usually called the **prover phase**. In this phase, a prover usually computes a valid assignment to the circuit. Depending on the use case, this valid assignment is then either directly used as constructive proof for proper circuit execution or is transferred as input to the proof generation algorithm of some zero-knowledge proof system, where the full-sized, non hiding constructive proof is processed into a succinct proof with various levels of zero knowledge.

Modern circuit languages and their associated compilers abstract over those two phases and provide a unified **interphase** to the developer, who then writes a single program that can be used in both phases.

To give the reader a clear, conceptual distinction between the two phases, `PAPER` keeps them separated. Code can be "brain-compiled" during the **setup-phase** in a pen-and-paper approach into visual circuits. Once a circuit is derived, it can be executed in a **prover phase** to generate a valid assignment. The valid assignment is then interpreted as a constructive proof for a knowledge claim in the associated language.

The Setup Phase In `PAPER`, the task of the setup phase is to compile code in the `PAPER` language into a visual representation of an algebraic circuit. Deriving the circuit from the code in a pen-and-paper style is what we call **brain-compiling**.

Given some statement description that adheres to the correct grammar, we start circuit development with an empty circuit, compile the main function first and then inductively compile all other functions as they are called during the process.

For every function we compile, we draw a box-node for every argument, every variable and every constant of that function. If the node represents a variable, we label it with that variable's name, and if it represents a constant, we label it with that constant's value. We group arguments into a subgraph labeled "inputs" and return values into a subgraph labeled "outputs". We then group everything into a subgraph and label that subgraph with the function's name.

After this is done, we have to do a consistency and type check for every occurrence of the assignment operator <==. We have to ensure that the expression on the right side of the operator is well defined and that the types of both side match.

Then we compile the right side of every occurrence of the assignment operator <==. If the right side is a constant or variable defined in this function, we draw a dotted line from the box-node that represents the left side of <== to the box node that represents the right side of the same operator. If the right side represents an argument of that function we draw a line from the box-node that represents the left side of <== to the box node that represents the right side of the same operator.

If the right side of the <== operator is a function, we look into our database, find its associated circuit and draw it. If no circuit is associated to that function yet, we repeat the compilation process for that function, drawing edges from the function's argument to its input nodes and from the functions output nodes to the nodes on the right side of <==.

During that process, edge labels are drawn according to the rules from 6.2.2.1. If the associated variable represents a private value, we use the W label to indicate a witness, and if it represents a public value, we use the I label to indicate an instance. check reference

Once this is done, we compile all occurring types in a function, by compiling the function of each type. We do this inductively until we reach the type of the base field. Circuits have no notion of types, only of field elements; hence, every type needs to be compiled to the field type in a sequence of compilation steps.

The compilation stops once we have inductively replaced all functions by their circuits. The result is a circuit that contains many unnecessary box nodes. In a final optimization step, all box nodes that are directly linked to each other are collapsed into a single node, and all box nodes that represent the same constants are collapsed into a single node.

Of course, PAPER's brain-compiler is not properly defined in any formal manner. Its purpose is to highlight important steps that real-world compilers undergo in their setup phases.

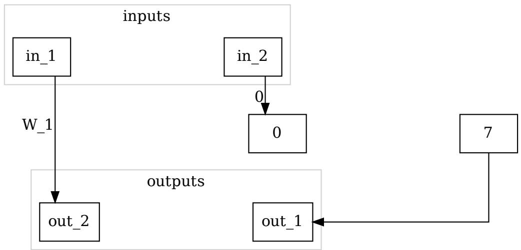
Example 125 (A trivial Circuit). To give an intuition of how to write and compile circuits in the PAPER language, consider the following statement description:

```
statement trivial_circuit {F:F_13} {
  fn main{F}(in1 : F, pub in2 : F) -> (F,F){
    let const outc1 : F = 0 ;
    let const inc1 : F = 7 ;
    let out1 : F ;
    let out2 : F ;
    out1 <== inc1;
    out2 <== in1;
    outc1 <== in2;
    return (out1, out2) ;
  }
}
```

To brain-compile this statement into an algebraic circuit with PAPER, we start with an empty circuit and evaluate function main, which is the only function in this statement.

We draw box-nodes for every argument, every constant and every variable of the function and label them with their names or values, respectively. Then we do a consistency and type check for every <== operator in the function. Since the circuit only wires inputs to outputs and all elements have the same type, the check is valid.

Then we evaluate the right side of the assignment operators. Since, in our case, the right side of each operator is not a function, we draw edges from the box-nodes on the right side to the associated box node on the left side. To label those edges, we use the general rules of algebraic circuits as defined in 6.2.2.1. According to those rules, every incoming edge of a sink node has a label and every outgoing edge of a source node has a label, if the node is labeled with a variable. Since nodes that represent constants are implicitly assumed to be private, and since the public specifier determines if an edge is labeled with W or I, we get the following circuit: check reference



The Prover Phase In `PAPER`, a so-called **prover phase** can be executed once the setup phase has generated a circuit image from its associated high-level code. This is done by executing the circuit while assigning proper values to all input nodes of the circuit. However, in contrast to most real-world compilers, `PAPER` does not tell the prover how to find proper input values to a given circuit. Real-world programing languages usually provide this data by computations that are done outside of the circuit.

Example 126. Consider the circuit from example 125. Valid assignments to this circuit are constructive proofs that the pair of inputs (S_1, S_2) is a point on the tiny-jubjub curve. However, the circuit does not provide a way to actually compute proper values for S_1 and S_2. Any real-world system therefore needs an auxiliary computation that provides those values. check reference

7.2 Common Programing concepts

In this section, we cover concepts that appear in almost every programming language, and see how they can be implemented in circuit compilers.

7.2.1 Primitive Types

Primitive data types like booleans, (unsigned) integers, or strings are the most basic building blocks one can expect to find in every general high-level programing language. In order to write statements as computer programs that compile into circuits, it is therefore necessary to implement primitive types as constraint systems, and define their associated operations as circuits.

In this section, we look at some common ways to achieve this. After a recapitulation of the atomic type of prime field elements, we start with an implementation of the boolean type and its associated boolean algebra as circuits. After that, we define unsigned integers based on the boolean type, and leave the implementation of signed integers as an exercise to the reader.

It should be noted, however, that while primitive data types in common programing languages (like C, Go, or Rust) have a one-to-one correspondence with objects in the computer's memory, this is not the case for most languages that compile into algebraic circuits. As we will see in the following paragraphs, common primitives like booleans or unsigned integers require many constraints and memory. Primitives different from the underlying field elements can be expensive.

The base-field type

Since both algebraic circuits and their associated rank-1 constraint systems are defined over a finite field, elements from that field are the atomic informational units in those models. In this sense, field elements $x \in \mathbb{F}$ are for algebraic circuits what bits are for computers.

In PAPER, we write F for this type and specify the actual field instance for every statement in curly brackets after the name of that statement. Two functions are associated to this type, which are induced by the **addition** and **multiplication** law in the field F. We write

$$\texttt{MUL} : \texttt{F} \times \texttt{F} \to \texttt{F} \,;\, (x,y) \mapsto \texttt{MUL}(x,y) \tag{7.1}$$

$$\texttt{ADD} : \texttt{F} \times \texttt{F} \to \texttt{F} \,;\, (x,y) \mapsto \texttt{ADD}(x,y) \tag{7.2}$$

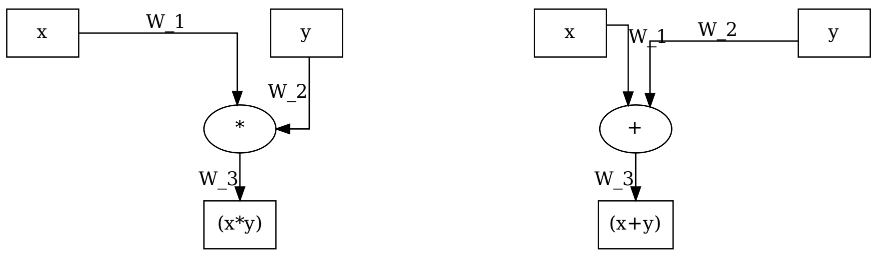
Circuit compilers have to compile these functions into algebraic gates, as explained in 6.2.2.2. check reference
Every other function has to be expressed in terms of them and proper wiring.

To represent addition and multiplication in the PAPER language, we define the following two functions:

```
fn MUL(x : F, y : F) -> (MUL(x,y):F){}
```

```
fn ADD(x : F, y : F) -> (ADD(x,y):F){}
```

The compiler then compiles every occurrence of the MUL or the ADD function into the following circuits:



Example 127 (Basic gates). To give an intuition of how a real-world compiler might transform addition and multiplication in algebraic expressions into a circuit, consider the following PAPER statement:

```
statement basic_ops {F:F_13} {
  fn main(in_1 : F, pub in_2 : F) -> (out_1:F, out_2:F){
    out_1 <== MUL(in_1,in_2) ;
    out_2 <== ADD(in_1,in_2) ;
  }
}
```

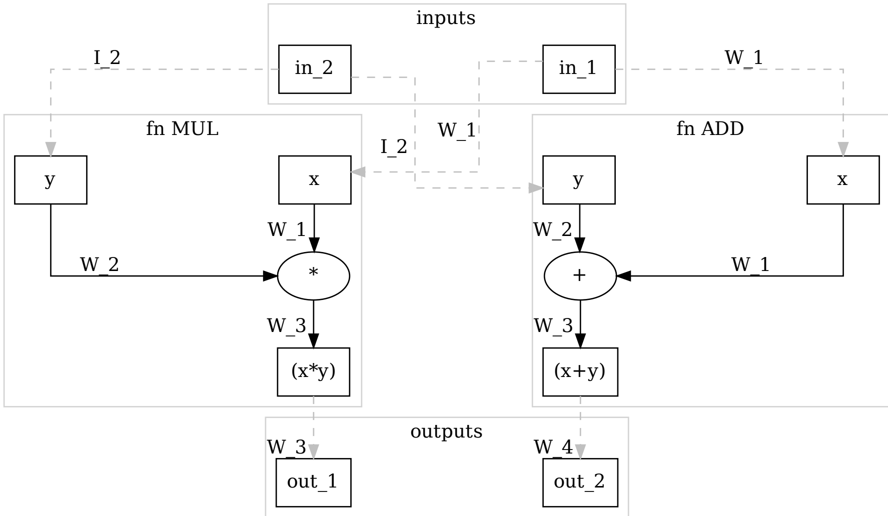
To compile this into an algebraic circuit, we start with an empty circuit and evaluate the function `main`, which is the only function in this statement.

We draw an inputs subgraph containing box-nodes for every argument of the function, and an outputs subgraph containing box-nodes for every factor in the return value. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

We check the validity of every expression on the right side of every <== operator including a type check. In our case, every variable is of the `field` type and hence the types match the types of the MUL as well as the ADD function and the type of the left sides of <== operators.

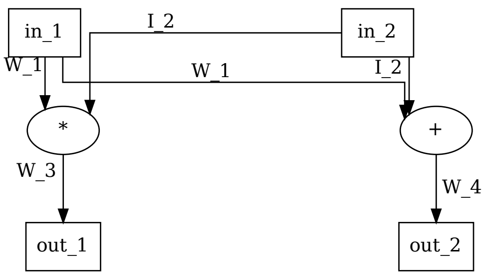
We evaluate the expressions on the right side of every <== operator inductively, replacing every occurrence of a function with a subgraph that represents its associated circuit.

According to PAPER, every occurrence of the `public` specifier overwrites the associate `private` default value. Using the appropriate edge labels we get:



Any real-world compiler might process its associated high-level language in a similar way, replacing functions, or gadgets by predefined associated circuits. This process is often followed by various optimization steps that try to reduce the number of constraints as much as possible.

In PAPER, we optimize this circuit by collapsing all box nodes that are directly connected to other box nodes, adhering to the rule that a variable's `public` specifier overwrites any `private` specifier. Reindexing edge labels, we get the following circuit as our pen and pencil compiler output:



Example 128 (3-factorization). Consider our 3-factorization problem from example 106 and the associated circuit $C_{3.fac\_zk}(\mathbb{F}_{13})$ we provided in equation 6.8. To understand the process of replacing high-level functions by their associated circuits inductively, we want define a PAPER statement that we brain-compile into an algebraic circuit equivalent to $C_{3.fac\_zk}(\mathbb{F}_{13})$:

check reference

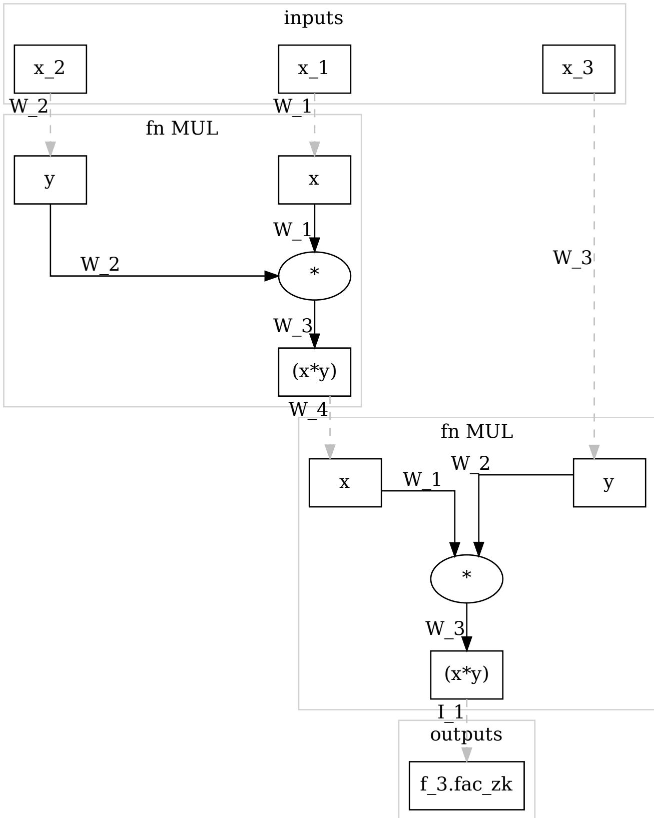
check reference

```
statement 3_fac_zk {F:F_13} {
  fn main(x_1 : F, x_2 : F, x_3 : F) -> (pub 3_fac_zk : F){
    f_3.fac_zk <== MUL( MUL( x_1 , x_2 ) , x_3 ) ;
  }
}
```

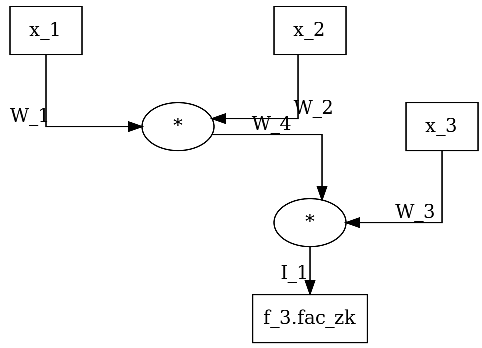
Using PAPER, we start with an empty circuit and then add 3 input nodes to the input subgraph as well as 1 output node to the output subgraph. All these nodes are decorated with the associated variable names. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

We check the validity of every expression on the right side of the single <== operator including a type check.

We evaluate the expressions on the right side of every <== operator inductively. We have two nested multiplication functions and we replace them by the associated multiplication circuits, starting with the most outer function. We get:

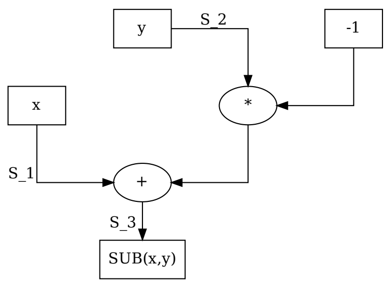


In a final optimization step, we collaps all box nodes directly connected to other box nodes, adhering to the rule that a variables `public` specifier overwrites any `private` specifier. Reindexing edge labels we get the following circuit:



The Subtraction Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field subtraction, despite the fact that subtraction is a native operation in every field.

High-level languages and their associated circuit compilers, therefore, need another way to deal with subtraction. To see how this can be achieved, recall that subtraction is defined by addition with the additive inverse, and that the inverse can be computed efficiently by multiplication with -1. A circuit for field subtraction is therefore given by



Using the general method from 6.2.1.1, the circuits associated rank-1 constraint system is given by: check reference

$$(S_1 + (-1) \cdot S_2) \cdot 1 = S_3 \tag{7.3}$$

Any valid assignment $\{S_1, S_2, S_3\}$ to this circuit therefore enforces the value S_3 to be the difference $S_1 - S_2$.

Real-world compilers usually provide a gadget or a function to abstract over this circuit such that programers can use subtraction as if it were native to circuits. In PAPER, we define the following subtraction function that compiles to the previous circuit:

```
fn SUB(x : F, y : F) -> (SUB(x,y) : F){
  constant c : F = -1 ;
  SUB <== ADD(x , MUL( y ,  c) );
}
```

In the setup phase of a statement, we compile every occurrence of the SUB function into an instance of its associated subtraction circuit, and edge labels are generated according to the rules from 6.2.2.1. check reference

The Inversion Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field inversion, despite the fact that inversion is a native operation in every field.

If the underlying field is a prime field, one approach would be to use Fermat's little theorem 3.3 to compute the multiplicative inverse inside the circuit. To see how this works, let $\mathbb{F}_p$ be the prime field. The multiplicative inverse x^{-1} of a field element $x \in \mathbb{F}$ with $x \neq 0$ is then given by $x^{-1} = x^{p-2}$, and computing x^{p-2} in the circuit therefore computes the multiplicative inverse.

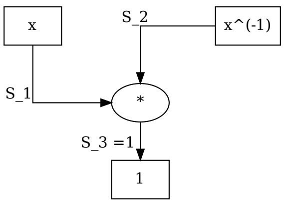
check reference

Unfortunately, real-world primes p are large and computing x^{p-2} by repeated multiplication of x with itself is infeasible. A "double and multiply" approach (as described in XXX) is faster, as it computes the power in roughly $log_2(p)$ steps, but still adds a lot of constraints to the circuit.

add reference

Computing inverses in the circuit makes no use of the fact that inversion is a native operation in any field. A more constraints friendly approach is therefore to compute the multiplicative inverse outside of the circuit and then only enforce correctness of the computation in the circuit.

To understand how this can be achieved, observe that a field element $y \in \mathbb{F}$ is the multiplicative inverse of a field element $x \in \mathbb{F}$ if and only if $x \cdot y = 1$ in $\mathbb{F}$. We can use this, and define a circuit that has two inputs, x and y, and enforces $x \cdot y = 1$. It is then guaranteed that y is the multiplicative inverse of x. The price we pay is that we can not compute y by circuit execution, but auxiliary data is needed to tell any prover which value of y is needed for a valid circuit assignment. The following circuit defines the constraint



Using the general method from 6.2.1.1, the circuit is transformed into the following rank-1 constraint system:

check reference

$$S_1 \cdot S_2 = 1 \tag{7.4}$$

Any valid assignment $\{S_1, S_2\}$ to this circuit enforces that S_2 is the multiplicative inverse of S_1, and, since there is no field element S_2 such that $0 \cdot S_2 = 1$, it also handles the fact that the multiplicative inverse of 0 is not defined in any field.

Real-world compilers usually provide a gadget or a function to abstract over this circuit, and those functions compute the inverse x^{-1} as part of their witness generation process. Programers then don't have to care about providing the inverse as auxiliary data to the circuit. In PAPER, we define the following inversion function that compiles to the previous circuit:

```
fn INV(x : F, y : F) -> (x_inv : F) {
  constant c : F = 1 ;
  c <== MUL( x ,  y ) ) ;
  x_inv <== y ;
}
```

As we see, this functions takes two inputs, the field value and its inverse. It therefore does not handle the computation of the inverse by itself. This is to keep PAPER as simple as possible.

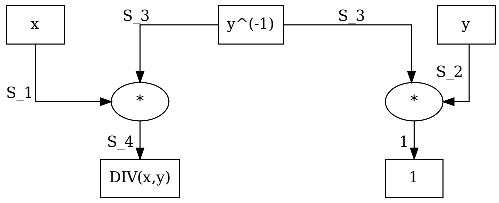
In the setup phase, we compile every occurrence of the INV function into an instance of the inversion circuit XXX, and edge labels are generated according to the rules from XXX.

add reference

add reference

The Division Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field division, despite the fact that division is a native operation in every field.

Implementing division as a circuit, we use the fact that division is multiplication with the multiplicative inverse. We therefore define division as a circuit using the inversion circuit and constraint system from the previous paragraph. Expensive inversion is computed outside of the circuit and then provided as circuit input. We get



Using the method from 6.2.1.1, we transform this circuit into the following rank-1 constraint system:

check reference

$$S_2 \cdot S_3 = 1$$
$$S_1 \cdot S_3 = S_4$$

Any valid assignment $\{S_1, S_2, S_3, S_4\}$ to this circuit enforces S_4 to be the field division of S_1 by S_2. It handles the fact that division by 0 is not defined, since there is no valid assignment in case $S_2 = 0$.

In PAPER, we define the following division function that compiles to the previous circuit:

```
fn DIV(x : F, y : F, y_inv : F) -> (DIV : F) {
  DIV <== MUL( x ,  INV( y, y_inv ) ) );
}
```

In the setup phase, we compile every occurrence of the binary INV operator into an instance of the inversion circuit.

Exercise 48. Let F be the field $\mathbb{F}_5$ of modular 5 arithmetics from example 13. Brain-compile the following PAPER statement into an algebraic circuit:

check reference

```
statement STUPID_CIRC {F: F_5} {
  fn foo(in_1 : F, in_2 : F)->(out_1 : F, out_2 : F,){
    constant c_1 : F = 3 ;
    out_1<== ADD( MUL( c_1 , in_1 ) , in_1 ) ;
    out_2<== INV( c_1 , in_2 ) ;
  } ;

  fn main(in_1 : F, in_2 ; F)->(out_1 : F, out_2 : TYPE_2){
    constant (c_1,c_2) : (F,F) = (3,2) ;
    (out_1,out_2) <== foo(in_1, in_2) ;
  } ;
}
```

Exercise 49. Consider the tiny-jubjub curve from example 66 and its associated circuit 125. Write a statement in PAPER that brain-compiles the statement into a circuit equivalent to the one derived in XXX, assuming that curve points are instances and every other assignment is a witness.

check reference

check reference

add reference

Exercise 50. Let $\mathtt{F} = \mathbb{F}_{13}$ be the modular 13 prime field and $x \in \mathtt{F}$ some field element. Define a statement in PAPER such that given instance x a field element $y \in \mathtt{F}$ is a witness for the statement if and only if y is the square root of x.

Brain-compile the statement into a circuit and derive its associated rank-1 constraint system. Consider the instance $x = 9$ and compute a constructive proof for the statement.

The boolean Type

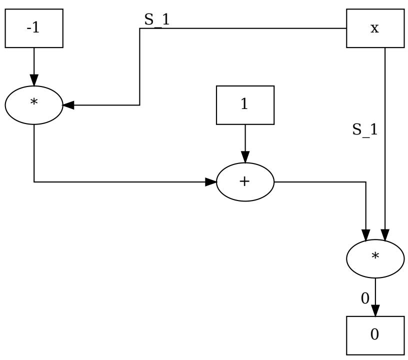
Booleans are a classical primitive type, implemented by virtually every higher programing language. It is therefore important to implement booleans in circuits. One of the most common ways to do this is by interpreting the additive and multiplicative neutral element $\{0,1\} \subset \mathbb{F}$ as the two boolean values such that 0 represents $false$ and 1 represents $true$. boolean operators like *and*, *or*, or *xor* are then expressible as algebraic computations inside $\mathbb{F}$.

Representing booleans this way is convenient, because the elements 0 and 1 are defined in any field. The representation is therefore independent of the actual field in consideration.

To fix boolean algebra notation, we write 0 to represent $false$ and 1 to represent $true$, and we write $\wedge$ to represent the boolean AND as well as $\vee$ to represent the boolean OR operator. The boolean NOT operator is written as $\neg$.

The boolean Constraint System To represent booleans by the additive and multiplicative neutral elements of a field, a constraint is required to actually enforce variables of boolean type to be either 1 or 0. In fact, many of the following circuits that represent boolean functions are only correct under the assumption that their input variables are constrained to be either 0 or 1. Not constraining boolean variables is a common problem in circuit design.

In order to constrain an arbitrary field element $x \in \mathbb{F}$ to be 1 or 0, the key observation is that the equation $x \cdot (1 - x) = 0$ has only two solutions 0 and 1 in any field. Implementing this equation as a circuit therefore generates the correct constraint:



Using the method from 6.2.1.1, we transform this circuit into the following rank-1 constraint system:

check reference

$$S_1 \cdot (1 - S_1) = 0$$

Any valid assignment $\{S_1\}$ to this circuit enforces S_1 to be either 0 or 1.

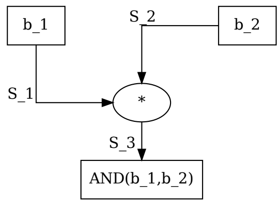
Some real-world circuit compilers (like ZOKRATES or BELLMAN) are typed, while others (like circom) are not. However, all of them have their way of dealing with the binary constraint. In PAPER, we define the following boolean type that compiles to the previous circuit:

```
type BOOL(b : BOOL) -> (x : F) {
    constant c1 : F = 0 ;
    constant c2 : F = 1 ;
    constant c3 : F = -1 ;
    c1 <== MUL( x ,  ADD( c2 , MUL( x , c3) ) ) );
    x <== b ;
}
```

In the setup phase of a statement, we compile every occurrence of a variable of boolean type into an instance of its associated boolean circuit.

The AND operator constraint system Given two field elements b_1 and b_2 from $\mathbb{F}$ that are constrained to represent boolean variables, we want to find a circuit that computes the logical **and** operator $AND(b_1,b_2)$ as well as its associated R1CS that enforces b_1, b_2, $AND(b_1,b_2)$ to satisfy the constraint system if and only if $b_1 \wedge b_2 = AND(b_1,b_2)$ holds true.

The key insight here is that, given three boolean constraint variables b_1, b_2 and b_3, the equation $b_1 \cdot b_2 = b_3$ is satisfied in $\mathbb{F}$ if and only if the equation $b_1 \wedge b_2 = b_3$ is satisfied in boolean algebra. The logical operator $\wedge$ is therefore implementable in $\mathbb{F}$ by field multiplication of its arguments and the following circuit computes the $\wedge$ operator in $\mathbb{F}$, assuming all inputs are restricted to be 0 or 1:



The associated rank-1 constraint system can be deduced from the general process 6.2.1.1 and consists of the following constraint:

check reference

$$S_1 \cdot S_2 = S_3 \tag{7.5}$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the $\wedge$ operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the $\wedge$-operator's circuit:

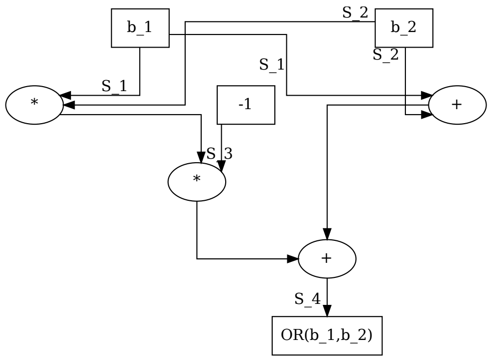
```
fn AND(b_1 : BOOL, b_2 : BOOL) -> AND(b_1,b_2) : BOOL{
  AND(b_1,b_2) <== MUL( b_1 ,  b_2) ;
}
```

In the setup phase of a statement, we compile every occurrence of the AND function into an instance of its associated $\wedge$-operator's circuit.

The OR operator constraint system Given two field elements b_1 and b_2 from $\mathbb{F}$ that are constrained to represent boolean variables, we want to find a circuit that computes the logical **or** operator $OR(b_1,b_2)$ as well as its associated R1CS that enforces b_1, b_2, $OR(b_1,b_2)$ to satisfy the constraint system if and only if $b_1 \vee b_2 = OR(b_1,b_2)$ holds true.

Assuming that three variables b_1, b_2 and b_3 are boolean constraint, the equation $b_1 + b_2 - b_1 \cdot b_2 = b_3$ is satisfied in $\mathbb{F}$ if and only if the equation $b_1 \vee b_2 = b_3$ is satisfied in boolean algebra. The logical operator $\vee$ is therefore implementable in $\mathbb{F}$ by the following circuit, assuming all inputs are restricted to be 0 or 1:

"constraints" or "constrained"?



The associated rank-1 constraint system can be deduced from the general process 6.2.1.1 and consists of the following constraints: check reference

$$S_1 \cdot S_2 = S_3$$
$$(S_1 + S_2 - S_3) \cdot 1 = S_4$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the $\vee$ operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the $\vee$-operator's circuit:

```
fn OR(b_1 : BOOL, b_2 : BOOL) -> OR(b_1,b_2) : BOOL{
  constant c1 : F = -1 ;
  OR(b_1,b_2) <== ADD(ADD(b_1,b_2),MUL(c1,MUL(b_1,b_2))) ;
}
```

In the setup phase of a statement, we compile every occurrence of the OR function into an instance of its associated $\vee$-operator's circuit.

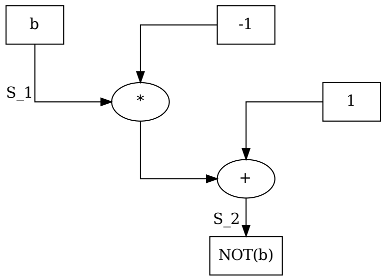
Exercise 51. Let $\mathbb{F}$ be a finite field and let b_1 as well as b_2 two boolean constraint variables from $\mathbb{F}$. Show that the equation $OR(b_1,b_2) = 1-(1-b_1)\cdot(1-b_2)$ holds true.

Use this equation to derive an algebraic circuit with ingoing variables b_1 and b_2 and outgoing variable $OR(b_1,b_2)$ such that b_1 and b_2 are boolean constraint and the circuit has a valid assignment, if and only if $OR(b_1,b_2) = b_1 \vee b_2$. "constraints" or "constrained"?

Use the technique from XXX to transform this circuit into a rank-1 constraint system and find its full solution set. Define a PAPER function that brain-compiles into the circuit. add reference

The NOT operator constraint system Given a field element b from $\mathbb{F}$ that is constrained to represent a boolean variable, we want to find a circuit that computes the logical **NOT** operator $NOT(b)$ as well as its associated R1CS that enforces b, $NOT(b)$ to satisfy the constraint system if and only if $\neg b = NOT(b)$ holds true.

Assuming that two variables b_1 and b_2 are boolean constraint, the equation $(1-b_1) = b_2$ is satisfied in $\mathbb{F}$ if and only if the equation $\neg b_1 = b_2$ is satisfied in boolean algebra. The logical operator $\neg$ is therefore implementable in $\mathbb{F}$ by the following circuit, assuming all inputs are restricted to be 0 or 1: "constraints" or "constrained"?



The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$(1 - S_1) \cdot 1 = S_2$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the $\neg$ operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the $\neg$-operator's circuit:

```
fn NOT(b : BOOL -> NOT(b) : BOOL{
  constant c1 = 1 ;
  constant c2 = -1 ;
  NOT(b_1) <== ADD( c1 , MUL( c2 , b) ) ;
}
```

In the setup phase of a statement, we compile every occurrence of the NOT function into an instance of its associated $\neg$-operator's circuit.

Exercise 52. Let $\mathbb{F}$ be a finite field. Derive the algebraic circuit and associated rank-1 constraint system for the following operators: NOR, XOR, NAND, EQU.

Modularity As we have seen in chapter 6,, both algebraic circuits and R1CS have a modularity property, and as we have seen in this section, all basic boolean functions are expressible in circuits. Combining those two properties, show that it is possible to express arbitrary boolean functions as algebraic circuits.

check references

This shows that the expressiveness of algebraic circuits and therefore rank-1 constraint systems is as general as the expressiveness of boolean circuits. An important implication is that the languages $L_{R1CS-SAT}$ and $L_{Circuit-SAT}$ as defined in 3, are as general as the famous language L_{3-SAT}, which is known to be $\mathcal{NP}$-complete.

check reference

Example 129. To give an example of how a compiler might construct complex boolean expressions in algebraic circuits from simple ones and how we derive their associated rank-1 constraint systems, let's look at the following PAPER statement:

```
statement BOOLEAN_STAT {F: F_p} {
  fn main(b_1:BOOL,b_2:BOOL,b_3:BOOL,b_4:BOOL )-> pub b_5:BOOL {
    b_5 <== AND( OR( b_1 , b_2) , AND( b_3 , NOT( b_4) ) ) ;
  } ;
}
```

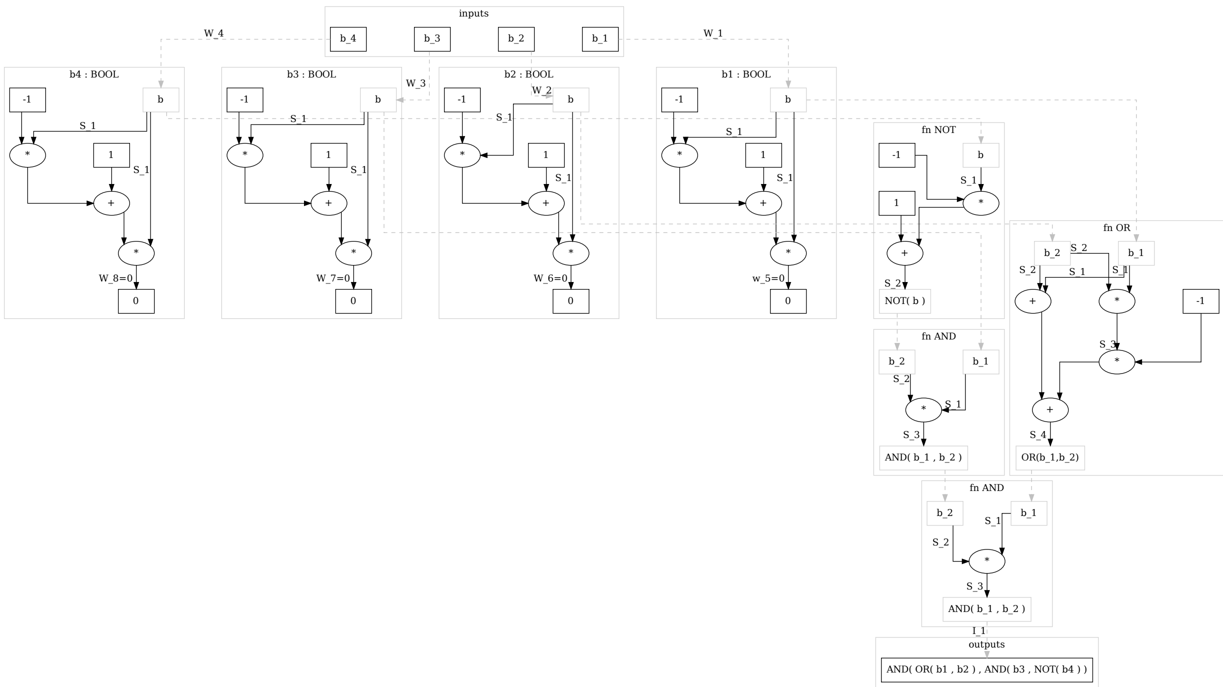
The code describes a circuit that takes four private inputs b_1, b_2, b_3 and b_4 of boolean type and computes a public output b_5 such that the following boolean expression holds true:

$$(b_1 \vee b_2) \wedge (b_3 \wedge \neg b_4) = b_5$$

During a setup-phase, a circuit compiler transforms this high-level language statement into a circuit and associated rank-1 constraint systems and hence defines a language $L_{BOOLEAN\_STAT}$.

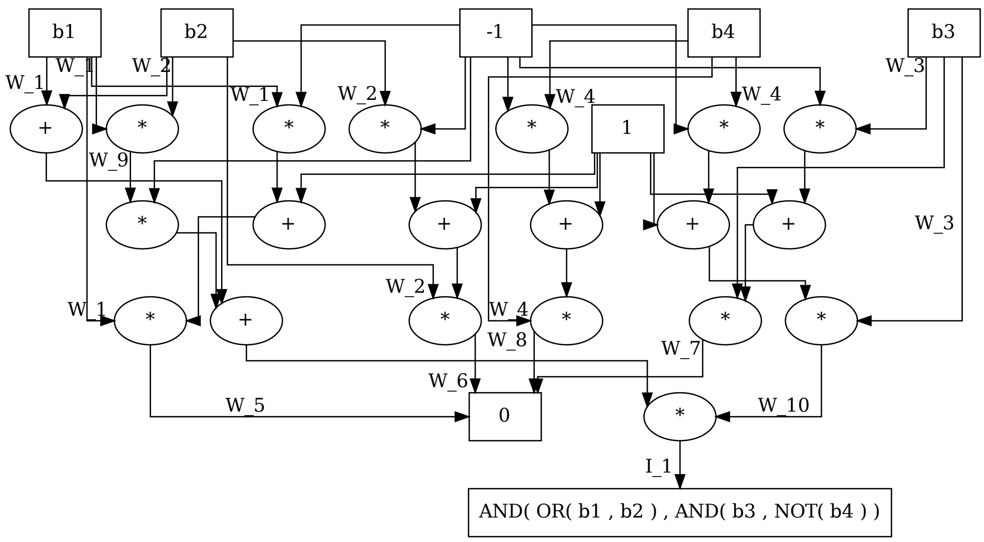
To see how this might be achieved, we use `PAPER` as an example to execute the setup-phase and compile `BOOLEAN_STAT` into a circuit. Taking the definition of the boolean constraint XXX as well as the definitions of the appropriate boolean operators into account, we get the following circuit:

add reference



can we rotate this by 90°?

Simple optimization then collapses all box-nodes that are directly linked and all box nodes that represent the same constants. After relabeling the edges, the following circuit represents the circuit associated to the `BOOLEAN_STAT` statement:



Given some public input I_1 from $\mathbb{F}_{13}$, a valid assignment to this circuit consists of private inputs W_1, W_2, W_3, W_4 from $\mathbb{F}_{13}$ such that the equation $I_1 = (W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4)$ holds true. In addition, a valid assignment also has to contain private inputs W_5, W_6, W_7, W_8, W_9 and W_{10}, which can be derived from circuit execution. The inputs W_5, ..., W_8 ensure that the first four private inputs are either 0 or 1 but not any other field element, and the others enforce the boolean operations in the expression.

To compute the associated R1CS, we can use the general method from 6.2.1.1 and look at every labeled outgoing edge not coming from a source node. We declare the edges coming from input nodes as well as the edge going to the single output node as public, and every other edge as private input. In this case we get: check reference

$$
\begin{aligned}
W_5 &: W_1 \cdot (1 - W_1) = 0 && \text{boolean constraints} \\
W_6 &: W_2 \cdot (1 - W_2) = 0 \\
W_7 &: W_3 \cdot (1 - W_3) = 0 \\
W_8 &: W_4 \cdot (1 - w_4) = 0 \\
W_9 &: W_1 \cdot W_2 = W_9 && \text{first OR-operator constraint} \\
W_{10} &: W_3 \cdot (1 - W_4) = W_{10} && \text{AND(.,NOT(.))-operator constraints} \\
I_1 &: (W_1 + W_2 - W_9) \cdot W_{10} = I_1 && \text{AND-operator constraints}
\end{aligned}
$$

The reason why this R1CS only contains a single constraint for the multiplication gate in the OR-circuit, while the general definition XXX requires two constraints, is that the second constraint in XXX only appears because the final addition gate is connected to an output node. In this case, however, the final addition gate from the OR-circuit is enforced in the left factor of the I_1 constraint. Something similar holds true for the negation circuit. add reference add reference

During a prover-phase, some public instance I_5 must be given. To compute a constructive proof for the statement of the associated languages with respect to instance I_5, a prover has to find four boolean values W_1, W_2, W_3 and W_4 such that

$$(W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4) = I_5$$

holds true. In our case neither the circuit nor the `PAPER` statement specifies how to find those values, and it is a problem that any prover has to solve outside of the circuit. This might or might not be true for other problems, too. In any case, once the prover found those values, they can execute the circuit to find a valid assignment.

To give a concrete example, let $I_1 = 1$ and assume $W_1 = 1$, $W_2 = 0$, $W_3 = 1$ and $W_4 = 0$. Since $(1 \vee 0) \wedge (1 \wedge \neg 0) = 1$, those values satisfy the problem and we can use them to execute the circuit. We get

$$
\begin{aligned}
W_5 &= W_1 \cdot (1 - W_1) = 0 \\
W_6 &= W_2 \cdot (1 - W_2) = 0 \\
W_7 &= W_3 \cdot (1 - W_3) = 0 \\
W_8 &= W_4 \cdot (1 - W_4) = 0 \\
W_9 &= W_1 \cdot W_2 = 0 \\
W_{10} &= W_3 \cdot (1 - W_4) = 1 \\
I_1 &= (W_1 + W_2 - W_9) \cdot W_{10} = 1
\end{aligned}
$$

A constructive proof of knowledge of a witness, for instance, $I_1 = 1$, is therefore given by the tuple $P = (W_5, W_6, W_7, W_8, W_9, W_{10}) = (0, 0, 0, 0, 0, 1)$.

Arrays

The `array` type represents a fixed-size collection of elements of equal type, each selectable by one or more indices that can be computed at run time during program execution.

Arrays are a classical type, implemented by many higher programing languages that compile to circuits or rank-1 constraint systems. However, most high-level circuit languages support **static** arrays, i.e., arrays whose length is known at compile time only.

The most common way to compile arrays to circuits is to transform any array of a given type `t` and size `N` into `N` circuit variables of type `N`. Arrays are therefore **syntactic sugar**, that is, parts of the formal language that makes the code easier for humans to read, which the compiler transforms into input nodes, much like any other variable. In `PAPER`, we define the following array type:

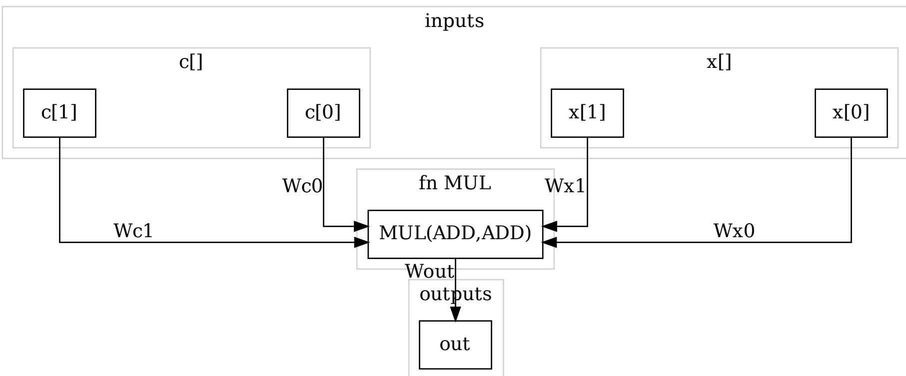
```
type <Name>: <Type>[N : unsigned] -> (Type,...) {
  return (<Name>[0],...)
}
```

In the setup phase of a statement, we compile every occurrence of an array of size `N` that contains elements of type `Type` into `N` variables of type `Type`.

Example 130. To give an intuition of how a real-world compiler might transform arrays into circuit variables, consider the following `PAPER` statement:

```
statement ARRAY_TYPE {F: F_5} {
  fn main(x: F[2])-> F {
    let constant c: F[2] = [2,4] ;
    let out:F <== MUL(ADD(x[1],c[0]),ADD(x[0],c[1])) ;
    return out ;
  } ;
}
```

During a setup phase, a circuit compiler might then replace any occurrence of the array type by a tuple of variables of the underlying type, and then use those variables in the circuit synthesis process instead. To see how this can be achieved, we use `PAPER` as an example. Abstracting over the sub-circuit of the computation, we get the following circuit:



The Unsigned Integer Type

Unsigned integers of size `N`, where `N` is usually a power of two, represent non-negative integers in the range $0 \ldots 2^N - 1$. They have a notion of addition, subtraction and multiplication, defined by modular 2^N arithmetics. If some `N` is given, we write `uN` for the associated type.

The uN Constraint System Many high-level circuit languages define the the various `uN` types as arrays of size `N`, where each element is of boolean type. This is similar to their representation on common computer hardware and allows for efficient and straightforward definition of common operators, like the various shift, or logical operators. shift

If some unsigned integer `N` is known at compile time in `PAPER`, we define the following `uN` type:

```
type uN -> BOOL[N] {
  let base2 : BOOL[N] <== BASE_2(uN) ;
  return base2 ;
}
```

To enfore an N-tuple of field elements $(b_0,\ldots,b_{N-1})$ to represent an element of type `uN` we therefore need N boolean constraints

$$
\begin{aligned}
S_0\cdot(1-S_0) &= 0\\
S_1\cdot(1-S_1) &= 0\\
&\cdots\\
S_{N-1}\cdot(1-S_{N-1}) &= 0
\end{aligned}
$$

In the setup phase of a statement, we compile every occurrence of the `uN` type by a size `N` array of boolean type. During a=the prover phase, actual elements of the uN type are first transformed into binary representation and then this binary representation is assigned to the boolean array that represents the `uN` type.

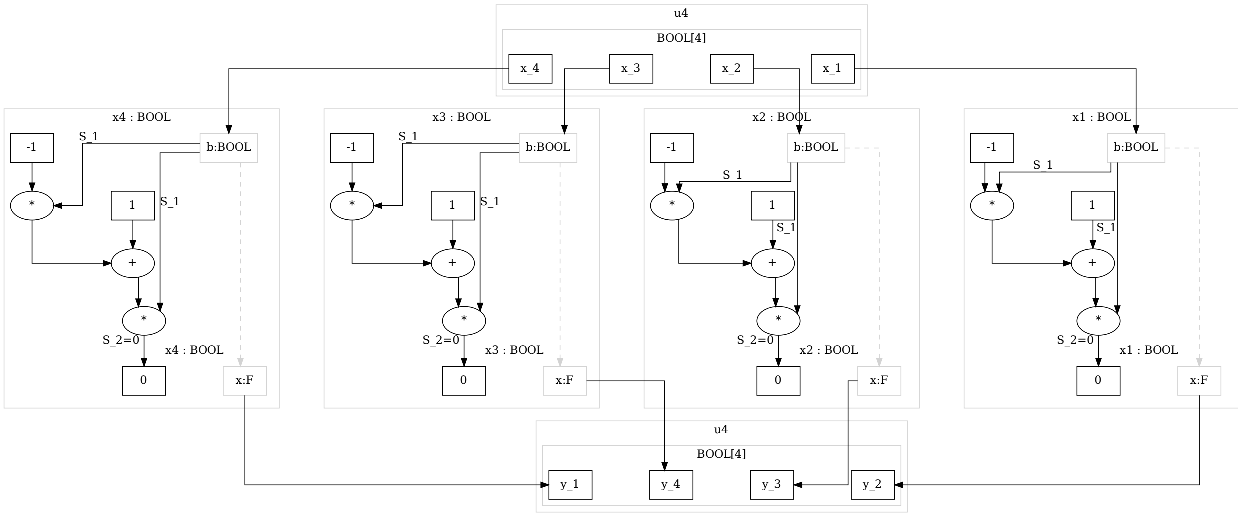
Remark 5. Representing the `uN` type as boolean arrays is conceptually clean and works over generic base fields. However, representing unsigned integers in this way requires a lot of space as every bit is represented as a field element and if the base field is large, those field elements require considerable space in hardware.

It should be noted that, in some cases, there is another, more space- and constraint-efficient approach for representing unsigned integers that can be used whenever the underlying base field is sufficiently large. To understand this, recall that addition and multiplication in a prime field $\mathbb{F}_p$ is equal to addition and multiplication of integers, as long as the sum or the product does not exceed the modulus p. It is therefore possible to represent the `uN` type inside the base-field type whenever N is small enough. In this case, however, care has to be taken to never overflow the modulus. It is also important to make sure that, in the case of subtraction, the subtrahend is never larger than the minuend.

Example 131. To give an intuition of how a real-world compiler might transform unsigned integers into circuit variables, consider the following `PAPER` statement:

```
statement RING_SHIFT{F: F_p, N=4} {
  fn main(x: uN)-> uN {
    let y:uN <== [x[1],x[2],x[3],x[0]] ;
    return y ;
  } ;
}
```

During the setup-phase, a circuit compiler might then replace any occurrence of the `uN` type by `N` variables of `boolean` type. Using the definition of booleans, each of these variables is then transformed into the `field` type and a boolean constraint system. To see how this can be achieved, we use `PAPER` as an example and get the following circuit:



During the prover phase, the function `main` is called with an actual input of `u4` type, say `x=14`. The high-level language then has to transform the decimal value 14 into its 4-bit binary representation $14_2 = (0,1,1,1)$ outside of the circuit. Then the array of field values $x[4] = [0,1,1,1]$ is used as an input to the circuit. Since all 4 field elements are either 0 or 1, the four boolean constraints are satisfiable and the output is an array of the four field elements $[1,1,1,0]$, which represents the `u4` element 7.

The Unigned Integer Operators Since elements of `uN` type are represented as boolean arrays, shift operators are implemented in circuits simply by rewiring the boolean input variables to the output variables accordingly.

Logical operators, like `AND`, `OR`, or `NOT` are defined on the `uN` type by invoking the appropriate boolean operators bitwise to every bit in the boolean array that represents the `uN` element.

Addition and multiplication can be represented similarly to how machines represent those operations. Addition can be implemented by first defining the **full adder** circuit and then combining N of these circuits into a circuit that adds two elements from the `uN` type.

Exercise 53. Let `F`$=\mathbb{F}_{13}$ and `N=4` be fixed. Define circuits and associated R1CS for the left and right bishift operators $x << 2$ as well as $x >> 2$ that operate on the `uN` type. Execute the associated circuit for $x : u4 = 11$.

Exercise 54. Let `F`$=\mathbb{F}_{13}$ and `N=2` be fixed. Define a circuit and associated R1CS for the addition operator `ADD` $:$ `F` $\times$ `F` $\to$ `F`. Execute the associated circuit to compute `ADD`$(2,7)$.

Exercise 55. Brain-compile the following `PAPER` code into a circuit and derive the associated R1CS.

```
statement MASK_MERGE {F:F_5, N=4} {
  fn main(pub a : uN, pub b : uN) -> F {
    let constant mask : uN = 10 ;
    let r : uN <== XOR(a,AND(XOR(a,b),mask)) ;
    return r ;
```

```
  }
}
```

Let L_{mask_merge} be the language defined by the circuit. Provide a constructive knowledge proof in L_{mask_merge} for the instance $I = (I_a, I_b) = (14, 7)$.

7.2.2 Control Flow

Most programing languages of the imperative of functional style have some notion of basic control structures to direct the order in which instructions are evaluated. Contemporary circuit compilers usually provide a single thread of execution and provide basic flow constructs that implement control flow in circuits.

The Conditional Assignment

Writing high-level code that compiles to circuits, it is often necessary to have a way for conditional assignment of values or computational output to variables.

One way to realize this in many programming languages is in terms of the conditional ternary assignment operator ? : that branches the control flow of a program according to some condition and then assigns the output of the computed branch to some variable.

```
variable = condition ? value_if_true : value_if_false
```

In this description, `condition` is a boolean expression and `value_if_true` as well as `value_if_false` are expressions that evaluate to the same type as `variable`.

In programming languages like Rust, another way to write the conditional assignment operator that is more familiar to many programmers is given by

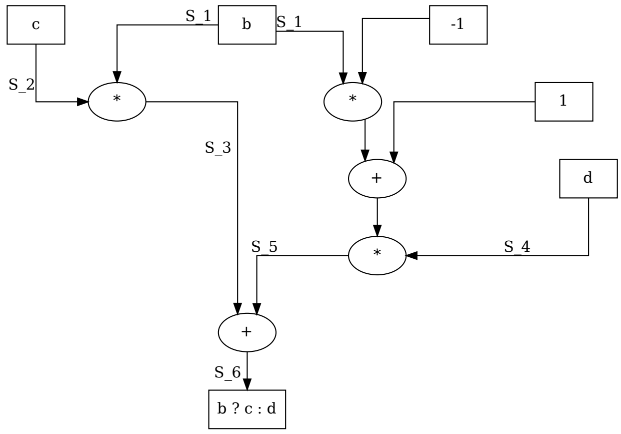
```
variable = if condition then {
  value_if_true
} else {
  value_if_false
}
```

In most programing languages, it is a key property of the ternary assignment operator that the expression `value_if_true` is only evaluated if `condition` evaluates to true and the expression `value_if_false` is only evaluated if `condition` evaluates to false. In fact, computer programs would turn out to be very inefficient if the ternary operator would evaluate both expressions regardless of the value of `condition`.

A simple way to implement conditional assignment operator as a circuit can be achieved if the requirement that only one branch of the conditional operator is executed is dropped. To see that, let b, c and d be field elements such that b is a boolean constraint. In this case, the following equation enforces a field element x to be the result of the conditional assignment operator:

$$x = b \cdot c + (1 - b) \cdot d \tag{7.6}$$

Expressing this equation in terms of the addition and multiplication operators from XXX, we can flatten it into the following algebraic circuit: add reference



Note that, in order to compute a valid assignment to this circuit, both S_2 as well as S_4 are necessary. If the inputs to the nodes c and d are circuits themself, both circuits need valid assignments and therefore have to be executed. As a consequence, this implementation of the conditional assignment operator has to execute all branches of all circuits, which is very different from the execution of common computer programs and contributes to the increased computational effort any prover has to invest, in contrast to the execution in other programing models.

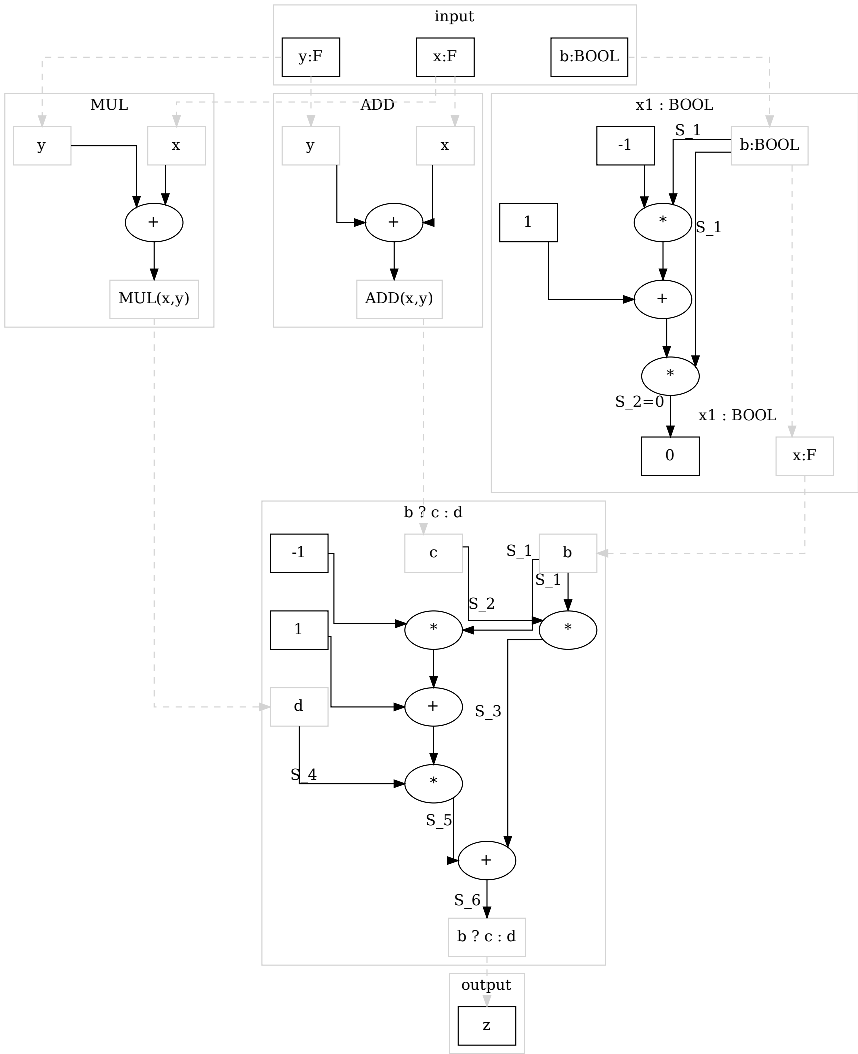
We can use the general technique from 6.2.1.1 to derive the associated rank-1 constraint system of the conditional assignment operator. We get the following: check reference

$$
\begin{aligned}
S_1 \cdot S_2 &= S_3 \\
(1 - S_1) \cdot S_4 &= S_5 \\
(S_3 + S_5) \cdot 1 &= S_6
\end{aligned}
$$

Example 132. To give an intuition of how a real-world circuit compiler might transform any high-level description of the conditional assignment operator into a circuit, consider the following `PAPER` code:

```
statement CONDITIONAL_OP {F:F_p} {
  fn main(x : F, y : F, b : BOOL) -> F {
    let z : F <== if b then {
      ADD(x,y)
    } else {
      MUL(x,y)
    } ;
    return z ;
  }
}
```

Brain-compiling this code into a circuit, we first draw box nodes for all input and output variables, and then transform the boolean type into the field type together with its associated constraint. Then we evaluate the assignments to the output variables. Since the conditional assignment operator is the top level function, we draw its circuit and then draw the circuits for both conditional expressions. We get the following:



Loops

In many programming languages, various loop control structures are defined that allow developers to execute expressions with a specified number of repetitions or arguments. In particular, it is often possible to implement unbounded loops like the loop structure give below, where the number of executions depends on execution inputs and is therefore unknown at compile time:

```
while true do { }
```

M: Add another example where the loop actually depends on the input.

Add example

In contrast to this, it should be noted that algebraic circuits and rank-1 constraint systems are not general enough to express arbitrary computation, but bounded computation only. As a consequence, it is not possible to implement unbounded loops, or loops with bounds that are unknown at compile time in those models. This can be easily seen since circuits are acyclic by definition, and implementing an unbounded loop as an acyclic graph requires a circuits of unbounded size.

However, circuits are general enough to express bounded loops, where the upper bound on its execution is known at compile time. Those loop can be implemented in circuits by enrolling the loop.

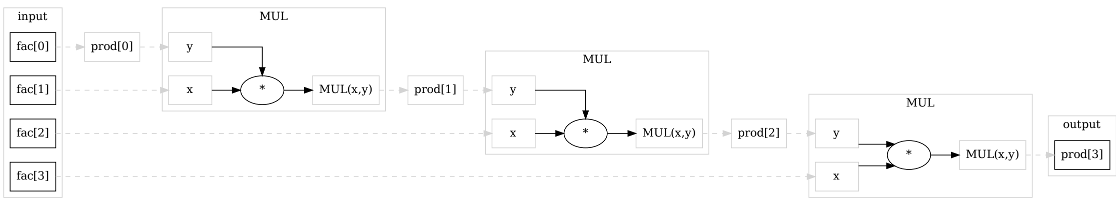
As a consequence, any programing language that compiles to algebraic circuits can only provide loop structures where the bound is a constant known at compile time. This implies that loops cannot depend on execution inputs, but on compile time parameters only.

Example 133. To give an intuition of how a real-world circuit compiler might transform any high-level description of a bounded `for` loop into a circuit, consider the following `PAPER` code:

```
statement FOR_LOOP {F:F_p, N: unsigned = 4} {
  fn main(fac : F[N]) -> F {
    let prod[N] : F ;
    prod[0] <== fac[0] ;
    for unsigned i in 1..N do [{
      prod[i] <== MUL(fac[i], prod[i-1]) ;
    }
    return prod[N] ;
  }
}
```

Note that, in a program like this, the loop counter `i` has no expression in the derived circuit. It is pure syntactic sugar, telling the compiler how to unroll the loop.

Brain-compiling this code into a circuit, we first draw box nodes for all input and output variables, noting that the loop counter is not represented in the circuit. Since all variables are of `field` type, we don't have to compile any type constraints. Then we evaluate the assignments to the output variables by unrolling the loop into 3 individual assignment operators. We get:



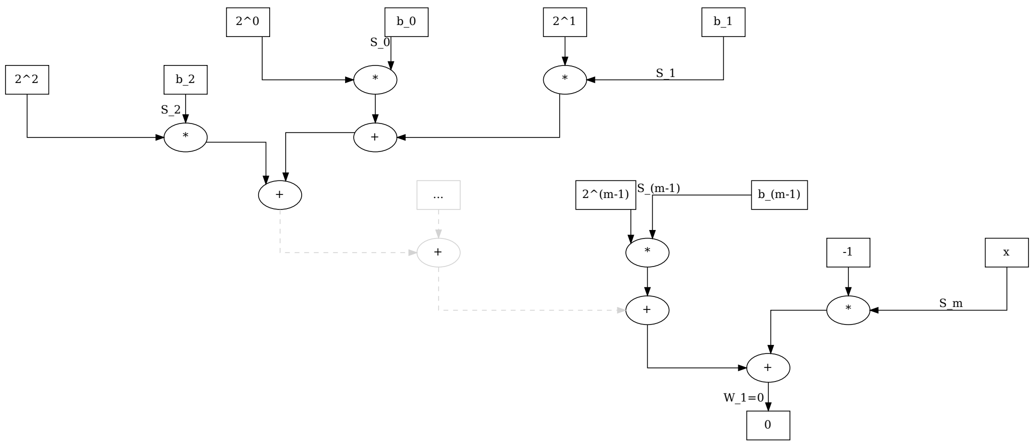
7.2.3 Binary Field Representations

In applications, is often necessary to enforce a binary representation of elements from the `field` type. To derive an appropriate circuit over a prime field $\mathbb{F}_p$, let $m = |p|_2$ be the smallest number of bits necessary to represent the prime modulus p. Then a bitstring $(b_0, \ldots, b_{m-1}) \in \{0,1\}^m$ is a binary representation of a field element $x \in \mathbb{F}_p$, if and only if

$$x = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_m \cdot 2^{m-1}$$

In this expression, addition and exponentiation is considered to be executed in $\mathbb{F}_p$, which is well defined since all terms 2^j for $0 \leq j < m$ are elements of $\mathbb{F}_p$. Note, however, that in contrast to the binary representation of unsigned integers $n \in \mathbb{N}$, this representation is not unique in general, since the modular p equivalence class might contain more than one binary representative.

Considering that the underlying prime field is fixed and the most significant bit of the prime modulus is m, the following circuit flattens equation XXX, assuming all inputs b_1, ..., b_m are of boolean type. add reference



Applying the general transformation rule to compute the associated rank-1 constraint systems, we see that we actually only need a single constraint to enforce some binary representation of any field element. We get

$$(S_0 \cdot 2^0 + S_1 \cdot 2^1 + \ldots + S_{m-1} \cdot 2^{m-1} - S_m) \cdot 1 = 0$$

Given an array `BOOL[N]` of `N` boolean constraint field elements and another field element x, the circuit enforces `BOOL[N]` to be one of the binary representations of x. If `BOOL[N]` is not a binary representation of x, no valid assignment and hence no solution to the associated R1CS can exists.

Example 134. Consider the prime field $\mathbb{F}_{13}$. To compute binary representations of elements from that field, we start with the binary representation of the prime modulus 13, which is $|13|_2 = (1,0,1,1)$ since $13 = 1\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2 + 1\cdot 2^3$. So $m = 4$ and we need up to 4 bits to represent any element $x \in \mathbb{F}_{13}$.

To see that binary representations are not unique in general, consider the element $2 \in \mathbb{F}_{13}$. It has the binary representations $|2|_2 = (0,1,0,0)$ and $|2|_2 = (1,1,1,1)$, since in $\mathbb{F}_{13}$ we have

$$2 = \begin{cases} 0\cdot 2^0 + 1\cdot 2^1 + 0\cdot 2^2 + 0\cdot 2^3 \\ 1\cdot 2^0 + 1\cdot 2^1 + 1\cdot 2^2 + 1\cdot 2^3 \end{cases}$$

This is because the unsigned integers 2 and 15 are both in the modular 13 remainder class of 2 and hence are both representatives of 2 in $\mathbb{F}_{13}$.

To see how circuit XXX works, we want to enforce the binary representation of $7 \in \mathbb{F}_{13}$. add reference Since $m = 4$ we have to enforce a 4-bit representation for 7, which is $(1,1,1,0)$, since $7 = 1\cdot 2^0 + 1\cdot 2^1 + 1\cdot 2^2 + 0\cdot 2^3$. A valid circuit assignment is therefore given by $(S_0, S_1, S_2, S_3, S_4) = (1,1,1,0,7)$ and, indeed, the assignment satifies the required 5 constraints including the 4 boolean constraints for S_0, ..., S_3:

$$\begin{aligned} 1\cdot(1-1) &= 0 && \text{// boolean constraints} \\ 1\cdot(1-1) &= 0 \\ 1\cdot(1-1) &= 0 \\ 0\cdot(1-0) &= 0 \\ (1+2+4+0-7)\cdot 1 &= 0 && \text{// binary rep. constraint} \end{aligned}$$

7.2.4 Cryptographic Primitives

In applications, it is often required to do cryptography in a circuit. To do this, basic cryptographic primitives like hash functions or elliptic curve cryptography needs to be implemented as circuits. In this section, we give a few basic examples of how to implement such primitives.

Twisted Edwards curves

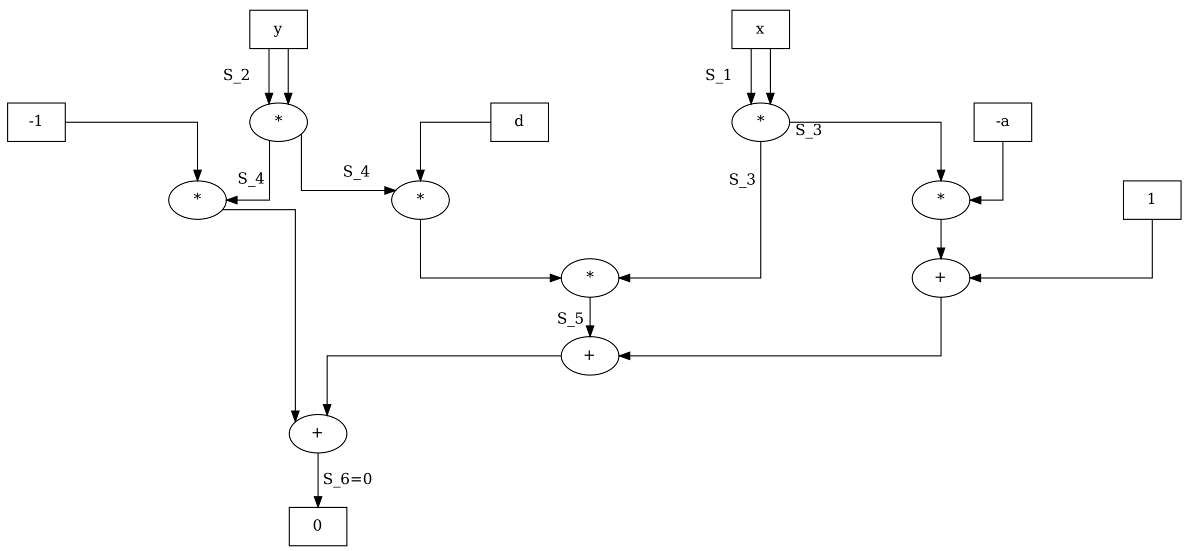
Implementing elliptic curve cryptography in circuits means to implement the field equation as well as the algebraic operations of an elliptic curve as circuits. To do this efficiently, the curve must be defined over the same base field as the field that is used in the circuit.

For efficiency reasons, it is advantageous to choose an elliptic curve such that that all required constraints and operations can be implement with as few gates as possible. Twisted Edwards curves are particularly useful for that matter, since their addition law is particularly simple and the same equation can be used for all curve points including the point at infinity. This simplifies the circuit a lot.

Twisted Edwards curve constraints As we have seen in section 5.1.3, a twisted Edwards curve over a finite field F is defined as the set of all pairs of points $(x,y) \in \mathbb{F}\times\mathbb{F}$ such that x and y satisfy the equation $a\cdot x^2+y^2 = 1+d\cdot x^2y^2$. As we have seen in example XXX, we can transform this equation into the following circuit:

check reference

add reference



The circuit enforces the two inputs of `field` type to satisfy the twisted Edwards curve equation and, as we know from example XXX, the associated rank-1 constraint system is given by:

add reference

$$
\begin{aligned}
S_1\cdot S_1 &= S_3\\
S_2\cdot S_2 &= S_4\\
(S_4\cdot 8)\cdot S_3 &= S_5\\
(12\cdot S_4+S_5+10\cdot S_3+1)\cdot 1 &= 0
\end{aligned}
$$

Exercise 56. Write the circuit and associated rank-1 constraint system for a Weierstraß curve of a given field $\mathbb{F}$.

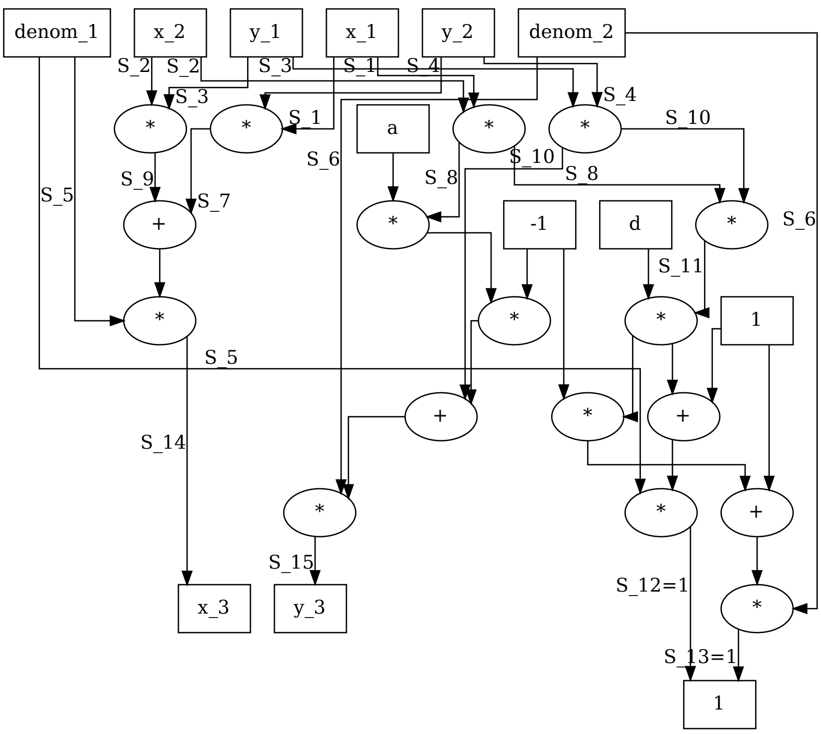
Twisted Edwards curve addition As we have seen in 5.1.3, a major advantage of twisted Edwards curves is the existence of an addition law that contains no branching and is valid for all curve points. Moreover, the neutral element is not "at infinity" but the actual curve point $(0,1)$. In fact, given two points (x_1, y_1) and (x_2, y_2) on a twisted Edwards curve, their sum is defined as

check reference

$$(x_3, y_3) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d \cdot x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a \cdot x_1 x_2}{1 - d \cdot x_1 x_2 y_1 y_2} \right)$$

We can use the division circuit from XXX to flatten this equation into an algebraic circuit. Inputs to the circuit are then not only the two curve points (x_1, y_1) and (x_2, y_2), but also the two denominators $denum_1 = 1 + d \cdot x_1 x_2 y_1 y_2$ as well as $denum_2 = 1 - d \cdot x_1 x_2 y_1 y_2$, which any prover needs to compute outside of the circuit. We get

add reference



Using the general technique from XXX to derive the associated rank-1 constraint system, we get the following result:

add reference

$$
\begin{aligned}
S_1 \cdot S_4 &= S_7 \\
S_1 \cdot S_2 &= S_8 \\
S_2 \cdot S_3 &= S_9 \\
S_3 \cdot S_4 &= S_{10} \\
S_8 \cdot S_{10} &= S_{11} \\
S_5 \cdot (1 + d \cdot S_{11}) &= 1 \\
S_6 \cdot (1 - d \cdot S_{11}) &= 1 \\
S_5 \cdot (S_9 + S_7) &= S_{14} \\
S_6 \cdot (S_{10} - a \cdot S_8) &= S_{15}
\end{aligned}
$$

Exercise 57. Let $\mathbb{F}$ be a field. Define a circuit that enforces field inversion for a point of a twisted Edwards curve over $\mathbb{F}$.

Chapter 8

Zero Knowledge Protocols

A so-called **zero-knowledge protocol** is a set of mathematical rules by which one party (usually called **the prover**) can convince another party (usually called **the verifier**) that a given statement is true, while not revealing any additional information apart from the truth of the statement.

As we have seen in chapter 6, given some language L and instance I, the knowledge claim "there is a witness W such that $(I;W)$ is a word in L" is constructively provable by providing W to the verifier. However, the challenge for a zero-knowledge protocol is to prove knowledge of a witness without revealing any information beyond its bare existence.

add reference

In this chapter, we look at various systems that exist to solve this task. We start with an introduction to the basic concepts and terminology in zero-knowledge proving systems and then introduce the so-called Groth_16 protocol as one of the most efficient systems. We plan to update this chapter with new inventions in future versions of this book.

8.1 Proof Systems

From an abstract point of view, a proof system is a set of rules which models the generation and exchange of messages between two parties: a prover and a verifier. Its task is to ascertain whether a given string belongs to a formal language or not.

Proof systems are often classified by certain trust assumptions and the computational capabilities of both parties. In its most general form, the prover usually possesses unlimited computational resources but cannot be trusted, while the verifier has bounded computation power but is assumed to be honest.

Proving the membership statement for some string is executed by the generation of certain messages that are sent between prover and verifier, until the verifier is convinced that the string is an element of the language in consideration.

To be more specific, let Σ be an alphabet, and let L be a formal language defined over Σ. Then a **proof system** for language L is a pair of probabilistic interactive algorithms (P,V), where P is called the **prover** and V is called the **verifier**.

Both algorithms are able to send messages to one another, and each algorithm has its own state, some shared initial state and access to the messages. The verifier is bounded to a number of steps which is polynomial in the size of the shared initial state, after which it stops and outputs either `accept` or `reject` indicating that it accepts or rejects a given string to be in L. In contrast, there are bounds on the computational power of the prover.

When the execution of the verifier algorithm stops the following conditions are required to hold:

- (Completeness) If the tuple $x \in \Sigma^*$ is a word in language L and both prover and verifier follow the protocol, the verifier outputs `accept`.
- (Soundness) If the tuple $x \in \Sigma^*$ is not a word in language L and the verifier follows the protocol, the verifier outputs `reject`, except with some small probability.

In addition, a proof system is called **zero-knowledge** if the verifier learns nothing about x other than $x \in L$.

The previous definition of proof systems is very general, and many subclasses of proving systems are known in the field. The type of languages that a proof system can support crucially depends on the abilities of the verifier (for example, whether it can make random choices) or on the nature and number of the messages that can be exchanged. If the system only requires to send a single message from the prover to the verifier, the proof system is called **non-interactive**, because no interaction other then sending the actual proof is required. In contrast, any other proof system is called **interactive**.

A proof system is usually called **succinct** if the size of the proof is shorter than the witness necessary to generate the proof. Moreover, a proof system is called **computationally sound** if soundness only holds under the assumption that the computational capabilities of the prover are polynomially bound. To distinguish general proofs from computationally sound proofs, the latter are often called **arguments**. Zero-knowledge, succinct, non-interactive arguments of knowledge claims are often abbreviated **zk-SNARKs**.

Example 135 (Constructive Proofs for Algebraic Circuits). To formalize our previous notion of constructive proofs for algebraic circuits, let $\mathbb{F}$ be a finite field, and let $C(\mathbb{F})$ be an algebraic circuit over $\mathbb{F}$ with associated language $L_{C(\mathbb{F})}$. A non-interactive proof system for $L_{C(\mathbb{F})}$ is given by the following two algorithms:

Given some instance I, the prover algorithm P uses its unlimited computational power to compute a witness W such that the pair $(I;W)$ is a valid assignment to $C(\mathbb{F})$ whenever the circuit is satisfiable for I. The prover then sends the constructive proof $(I;W)$ to the verifier.

On receiving a message $(I;W)$, the verifier algorithm V assigns the constructive proof $(I;W)$ to circuit $C(\mathbb{F})$, and decides whether the assignment is valid by executing all gates in the circuit. The runtime is polynomial in the number of gates. If the assignment is valid, the verifier returns `accepts`, if not, it returns `reject`.

To see that this proof system has the completeness and soundness properties, let $C(\mathbb{F})$ be a circuit of the field $\mathbb{F}$, and let I be an instance. The circuit may or may not have a witness W such that $(I;W)$ is a valid assignment to $C(\mathbb{F})$.

If no W exists, I is not part of any word in $L_{C(\mathbb{F})}$, and there is no way for P to generate a valid assignment. If follows that the verifier will not accept any claimed proof sent by P, which implies that the system has **soundness**.

If, on the other hand, W exists and P is honest, P can use its unlimited computational power to compute W and send $(I;W)$ to V, which V will accept in polynomial time. This implies that the system has **completeness**.

The system is non-interactive because the prover only sends a single message to the verifier, which contains the proof itself. Because in this simple system the witness itself is the proof, the proof system is **not** succinct.

8.2 The "Groth16" Protocol

In chapter 6, we have introduced algebraic circuits, their associated rank-1 constraint systems and their induced quadratic arithmetic programs. These models define formal languages, and check reference associated memberships and knowledge claims can be constructively proofed by executing the circuit to compute a solution to its associated R1CS. The solution can then be transformed into a polynomial such that the polynomial is divisible by another polynomial if and only if the solution is correct.

In **?**, Jens Groth provides a method that can transform those proofs into zero-knowledge succinct non-interactive arguments of knowledge. Assuming that the pairing groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, b)$ are given, the arguments are of constant size and consist of 2 elements from G_1 and a single element from $\mathbb{G}_2$, regardless of the size of the witness. They are zero-knowledge in the sense that the verifier learns nothing about the witness beside the fact that the instance-witness pair is a proper word in the language of the problem.

check reference

Verification is non-interactive and needs to compute a number of exponentiations proportional to the size of the instance, together with 3 group pairings in order to check a single equation.

The generated argument is perfectly zero-knowledge has perfect completeness and soundness in the generic bilinear group model, assuming the existence of a trusted third party that executes a preprocessing phase to generate a common reference string and a simulation trapdoor. This party must be trusted to delete the simulation trapdoor, since everyone in possession of it can simulate proofs.

common reference string

simulation trapdoor

To be more precise, let R be a rank-1 constraint system defined over some finite field $\mathbb{F}_r$. Then **Groth_16 parameters** for R are given by the following set:

$$\texttt{Groth\_16} - \texttt{Param}(R) = (r, \mathbb{G}_1, \mathbb{G}_2, e(\cdot,\cdot), g_1, g_2) \tag{8.1}$$

In the equation above, $\mathbb{G}_1$ and $\mathbb{G}_2$ are finite cyclic groups of order r, g_1 is a generator of $\mathbb{G}_1$, g_2 is a generator of $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate, bilinear pairing for some target group $\mathbb{G}_T$. In real-world applications, the parameter set is usually agreed on in advance.

Given some Groth_16 parameters, a **Groth_16 protocol** is then a quadruple of probabilistic polynomial algorithms $(\textsc{Setup}, \textsc{Prove}, \textsc{Vfy}, \textsc{Sim})$ such that the following conditions hold:

- (Setup-Phase): $(CRS, \tau) \leftarrow \texttt{Setup}(R)$: Algorithm `Setup` takes the R1CS R as input and computes a common reference string CRS and a simulation trapdoor τ.

- (Prover-Phase): $\pi \leftarrow Prove(R, CRS, I, W)$: Given a constructive proof $(I;W)$ for R, algorithm `Prove` takes the R1CS R, the common reference string CRS and the constructive proof (I,W) as input and computes an zk-SNARK π.

- Verify: $\{\texttt{accept}, \texttt{reject}\} \leftarrow Vfy(R, CRS, I, \pi)$: Algorithm `Vfy` takes the R1CS R, the common reference string CRS, the instance I and the zk-SNARK π as input and returns `reject` or `accept`.

- $\pi \leftarrow Sim(R, \tau, CRS, I)$: Algorithm `Sim` takes the R1CS R, the common reference string CRS, the simulation trapdoor τ and the instance I as input and returns a zk-SNARK π.

We will explain these algorithms together with detailed examples in the remainder of this section.

Assuming a trusted third party for the setup, the protocol is able to compute a zk-SNARK from a constructive proof for R, provided that r is sufficiently large, and, in particular, larger than the number of constraints in the associated R1CS.

Example 136 (The 3-Factorization Problem). Consider the 3-factorization problem from 106 and its associated algebraic circuit and rank-1 constraint system from 6.8. In this example,

check reference

check reference

we want to agree on a parameter set $(R, r, \mathbb{G}_1, \mathbb{G}_2, e(\cdot,\cdot), g_1, g_2)$ in order to use the Groth_16 protocol for our 3-factorization problem.

To find proper parameters, first observe that the circuit XXX, as well as its associated R1CS $R_{3.fac\_zk}$ ex:3-factorization-r1cs and the derived QAP 6.14, are defined over the field $\mathbb{F}_{13}$. We therefore have $r = 13$ and need pairing groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order 13.

add reference

check reference

We know from 5.4 that the moon-math curve `BLS6_6` has two subgroups $\mathbb{G}_1[13]$ and $\mathbb{G}_2[13]$, which are both of order 13. The associated Weil pairing b (5.45) is a proper bilinear map. We therefore choose those groups and the Weil pairing together with the generators $g_1 = (13, 15)$ and $g_2 = (7v^2, 16v^3)$ of $\mathbb{G}_1[13]$ and $\mathbb{G}_2[13]$, as a parameter:

check reference

check reference

$$\texttt{Groth\_16}-\texttt{Param}(R_{3.fac\_zk}) = (r, \mathbb{G}_1[13], \mathbb{G}_2[13], e(\cdot,\cdot), (13, 15), (7v^2, 16v^3))$$

It should be noted that our choice is not unique. Every pair of finite cyclic groups of order 13 that has a proper bilinear pairing qualifies as a Groth_16 parameter set. The situation is similar to real-world applications, where SNARKs with equivalent behavior are defined over different curves, used in different applications.

The Setup Phase To generate zk-SNARKs from constructive knowledge proofs in the Groth16 protocol, a preprocessing phase is required. This has to be executed a single time for every rank-1 constraint system and any associated quadratic arithmetic program. The outcome of this phase is a common reference string that prover and verifier need in order to generate and verify the zk-SNARK. In addition, a simulation trapdoor is produced that can be used to simulate proofs.

To be more precise, let L be a language defined by some rank-1 constraint system R such that a constructive proof of knowledge for an instance $(I_1, \ldots, I_n)$ in L consists of a witness $(W_1, \ldots, W_m)$. Let $QAP(R) = \left\{ T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m} \right\}$ be a quadratic arithmetic program associated to R, and let $\{\mathbb{G}_1, \mathbb{G}_2, e(\cdot,\cdot), g_1, g_2, \mathbb{F}_r\}$ be the set of Groth_16 parameters.

The setup phase then samples 5 random, inverible elements α, β, γ, δ and s from the scalar field $\mathbb{F}_r$ of the protocol and outputs the **simulation trapdoor** τ: "invariable"?

$$\tau = (\alpha, \beta, \gamma, \delta, s) \tag{8.2}$$

In addition, the setup phase uses those 5 random elements together with the two generators g_1 and g_2 and the quadratic arithmetic program to generate a **common reference string** $CRS_{QAP} = (CRS_{\mathbb{G}_1}, CRS_{\mathbb{G}_2})$ of language L:

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{c} g_1^{\alpha}, g_1^{\beta}, g_1^{\delta}, \left(g_1^{s^j}, \ldots\right)_{j=0}^{deg(T)-1}, \left(g_1^{\frac{\beta \cdot A_j(s) + \alpha \cdot B_j(s) + C_j(s)}{\gamma}}, \ldots\right)_{j=0}^{n} \\ \left(g_1^{\frac{\beta \cdot A_{j+n}(s) + \alpha \cdot B_{j+n}(s) + C_{j+n}(s)}{\delta}}, \ldots\right)_{j=1}^{m}, \left(g_1^{\frac{s^j \cdot T(s)}{\delta}}, \ldots\right)_{j=0}^{deg(T)-2} \end{array} \right\}$$

$$CRS_{\mathbb{G}_2} = \left\{ g_2^{\beta}, g_2^{\gamma}, g_2^{\delta}, \left(g_2^{s^j}, \ldots\right)_{j=0}^{deg(T)-1} \right\}$$

Common reference strings depend on the simulation trapdoor, and are therefore not unique to the problem. Any language can have more than one common reference string. The size of a common reference string is linear in the size of the instance and the size of the witness.

If a simulation trapdoor $\tau = (\alpha, \beta, \gamma, \delta, s)$ is given, we call the element s a **secret evaluation point** of the protocol, because if $\mathbb{F}_r$ is the scalar field of the finite cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$, then a key feature of any common reference string is that it provides data to compute the evaluation of any polynomial $P \in \mathbb{F}_r[x]$ of degree $deg(P) < deg(T)$ at the point s in the exponent of the generator g_1 or g_2, without knowing s.

To be more precise, let s be the secret evaluation point and let $P(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + \ldots a_k \cdot x^k$ be a polynomial of degree $k < deg(T)$ with coefficients in $\mathbb{F}_r$. Then we can compute $g_1^{P(s)}$ without knowing what the actual value of s is:

$$
\begin{aligned}
g_1^{P(s)} &= g_1^{a_0\cdot s^0 + a_1 \cdot s^1 + \ldots a_k \cdot s^k} \\
&= g_1^{a_0 \cdot s^0} \cdot g_1 a_1 \cdot s^1 \cdot \ldots \cdot g_1^{a_k \cdot s^k} \\
&= \left(g_1^{s^0}\right)^{a_0} \cdot \left(g_1^{s^1}\right)^{a_1} \cdot \ldots \cdot \left(g_1^{s^k}\right)^{a_k}
\end{aligned}
$$

In this expression, all group points $g_1^{s^j}$ are part of the common reference string, hence, they can be used to compute the result. The same holds true for the evaluation of $g_2^{P(s)}$, since the $\mathbb{G}_2$ part of the common reference string contains the points $g_2^{s^j}$.

In real-world applications, the simulation trapdoor is often called the **toxic waste** of the setup-phase, while a common reference string is also-called the pair of **prover and verifier key**.

explain why

In order to make the protocol secure, the setup needs to be executed in a way that guarantees that the simulation trapdoor is deleted. Anyone in possession of it can generate arguments without knowledge of a constructive proof. The most simple approach to achieving deletion of the toxic waste is by a so-called **trusted third party**, where the trust assumption is that that the party generates the common reference string precisely as defined and deletes the simulation backdoor afterwards.

However, as trusted third parties are not easy to find, more sophisticated protocols exist in real-world applications. They execute the setup phase as a multi party computation, where the proper execution can be publicly verified and the simulation trapdoor is deleted if at least one participant deletes their individual contribution to the randomness. Each participant only possesses a fraction of the simulation trapdoor, so it can only be recovered if all participants collude and share their fraction.

Example 137 (The 3-factorization Problem). To see how the setup phase of a Groth_16 zk-SNARK can be computed, consider the 3-factorization problem from 106 and the parameters from page 191. As we have seen in 6.14, an associated quadratic arithmetic program is given as follows:

4 examples have the same title. Change it to be distinct

check reference

add reference

check reference

$$
\begin{aligned}
QAP(R_{3.fac\_zk}) = \{&x^2 + x + 9, \\
&\{0,0,6x+10,0,0,7x+4\}, \{0,0,0,6x+10,7x+4,0\}, \{0,7x+4,0,0,0,6x+10\}\}
\end{aligned}
$$

To transform this QAP into a common reference string, we choose the field elements $\alpha = 6$, $\beta = 5$, $\gamma = 4$, $\delta = 3$, $s = 2$ from $\mathbb{F}_{13}$. In real-world applications, it is important to sample those values randomly from the scalar field, but in our approach, we choose those non-random values to make them more memorizable, which helps in pen-and-paper computations. Our simulation trapdoor is then given as follows:

$$\tau = (6,5,4,3,2)$$

We keep this secret in order to simulate proofs later on, butwe are careful though to hide τ from anyone who hasn’t read this book. Then we instantiate the common reference string XXX

add reference

from those values. Since our groups are subgroups of the `BLS6_6` elliptic curve, we use scalar product notation instead of exponentiation.

To compute the $\mathbb{G}_1$ part of the common reference string, we use the logarithmic order of the group $\mathbb{G}_1$ XXX, the generator $g_1 = (13,15)$, as well as the values from the simulation backdoor. Since $deg(T) = 2$, we get the following:

add reference

$$
\begin{aligned}
[\alpha]g_1 &= [6](13,15) = (27,34)\\
[\beta]g_1 &= [5](13,15) = (26,34)\\
[\delta]g_1 &= [3](13,15) = (38,15)
\end{aligned}
$$

To compute the rest of the $\mathbb{G}_1$ part of the common reference string, we expand the indexed tuples and insert the secret random elements from the simulation backdoor. We get the following:

$$
\begin{aligned}
\Big([s^j]g_1,\ldots\Big)_{j=0}^{1} &= \Big([2^0](13,15),[2^1](13,15)\Big)\\
&= \Big((13,15),(33,34)\Big)\\
\Big([\frac{\beta A_j(s)+\alpha B_j(s)+C_j(s)}{\gamma}]g_1,\ldots\Big)_{j=0}^{1} &= \Big([\frac{5A_0(2)+6B_0(2)+C_0(2)}{4}](13,15),\\
&\quad [\frac{5A_1(2)+6B_1(2)+C_1(2)}{4}](13,15)\Big)\\
\Big([\frac{\beta A_{j+n}(s)+\alpha B_{j+n}(s)+C_{j+n}(s)}{\delta}]g_1,\ldots\Big)_{j=1}^{4} &= \Big([\frac{5A_2(2)+6B_2(2)+C_2(2)}{3}](13,15),\\
&\quad [\frac{5A_3(2)+6B_3(2)+C_3(2)}{3}](13,15),\\
&\quad [\frac{5A_4(2)+6B_4(2)+C_4(2)}{3}](13,15),\\
&\quad [\frac{5A_5(2)+6B_5(2)+C_6(2)}{3}](13,15)\Big)\\
\Big([\frac{s^j\cdot T(s)}{\delta})]g_1\Big)_{j=0}^{0} &= \Big([\frac{2^0\cdot T(2)}{3}](13,15)\Big)
\end{aligned}
$$

To compute the curve points on the right side of these expressions, we need the polynomials from the associated quadratic arithmetic program and evaluate them on the secret point $s = 2$.

Since $4^{-1} = 10$ and $3^{-1} = 9$ in $\mathbb{F}_{13}$, we get the following:

$$
\begin{aligned}
[\frac{5A_0(2)+6B_0(2)+C_0(2)}{4}](13,15) =& [(5\cdot 0+6\cdot 0+0)\cdot 10](13,15) = [0](13,14) \\
& \mathcal{O} \\
[\frac{5A_1(2)+6B_1(2)+C_1(2)}{4}](13,15) =& [(5\cdot 0+6\cdot 0+(7\cdot 2+4))\cdot 10](13,15) = [11](13,15) = \\
& (33,9) \\
[\frac{5A_2(2)+6B_2(2)+C_2(2)}{3}](13,15) =& [(5\cdot(6\cdot 2+10)+6\cdot 0+0)\cdot 9](13,15) = [2](13,15) = \\
& (33,34) \\
[\frac{5A_3(2)+6B_3(2)+C_3(2)}{3}](13,15) =& [(5\cdot 0+6\cdot(6\cdot 2+10)+0)\cdot 9](13,15) = [5](13,15) = \\
& (26,34) \\
[\frac{5A_4(2)+6B_4(2)+C_4(2)}{3}](13,15) =& [(5\cdot 0+6\cdot(7\cdot 2+4)+0)\cdot 9](13,15) = [10](13,15) = \\
& (38,28) \\
[\frac{5A_5(2)+6B_5(2)+C_5(2)}{3}](13,15) =& [(5\cdot(7\cdot 2+4)+6\cdot 0+0)\cdot 9](13,15) = [4](13,15) = \\
& (35,28) \\
[\frac{2^0\cdot T(2)}{3}](13,15) =& [1\cdot(2^2+2+9)\cdot 9](13,15) = [5](13,15) = \\
& (26,34)
\end{aligned}
$$

Putting all those values together, we see that the $\mathbb{G}_1$ part of the common reference string is given by the following set of 12 points from the `BLS6_6` 13-torsion group $\mathbb{G}_1$:

$$
CRS_{\mathbb{G}_1} = \left\{ \begin{array}{c} (27,34),(26,34),(38,15),\Big((13,15),(33,34)\Big),\Big(\mathcal{O},(33,9)\Big) \\ \Big((33,34),(26,34),(38,28),(35,28)\Big),\Big((26,34)\Big) \end{array} \right\}
$$

To compute the $\mathbb{G}_2$ part of the common reference string, we use the logarithmic order of the group $\mathbb{G}_2$ XXX, the generator $g_2 = (7v^2, 16v^3)$, as well as the values from the simulation backdoor. Since $deg(T) = 2$, we get the following: add reference

$$
\begin{aligned}
[\beta]g_2 &= [5](7v^2,16v^3) = (16v^2,28v^3) \\
[\gamma]g_2 &= [4](7v^2,16v^3) = (37v^2,27v^3) \\
[\delta]g_2 &= [3](7v^2,16v^3) = (42v^2,16v^3)
\end{aligned}
$$

To compute the rest of the $\mathbb{G}_2$ part of the common reference string, we expand the indexed tuple and insert the secret random elements from the simulation backdoor. We get the following:

$$
\begin{aligned}
\Big([s^j]g_2,\dots\Big)_{j=0}^{1} &= \Big([2^0](7v^2,16v^3),[2^1](7v^2,16v^3)\Big) \\
&= \Big((7v^2,16v^3),(10v^2,28v^3)\Big)
\end{aligned}
$$

Putting all these values together, we see that the $\mathbb{G}_2$ part of the common reference string is given by the following set of 5 points from the `BLS6_6` 13-torsion group $\mathbb{G}_2$:

$$
CRS_{\mathbb{G}_2} = \Big\{(16v^2,28v^3),(37v^2,27v^3),(42v^2,16v^3),\Big((7v^2,16v^3),(10v^2,28v^3)\Big)\Big\}
$$

Given the simluation trapdoor τ and the quadratic arithmetic program 6.14, the associated common reference string of the 3-factorization problem is as follows: check reference

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{c} (27,34),(26,34),(38,15),\Big((13,15),(33,34)\Big),\Big(\mathcal{O},(33,9)\Big) \\ \Big((33,34),(26,34),(38,28),(35,28)\Big),\Big((26,34)\Big) \end{array} \right\}$$

$$CRS_{\mathbb{G}_2} = \Big\{(16v^2,28v^3),(37v^2,27v^3),(42v^2,16v^3),\Big(7v^2,16v^3),(10v^2,28v^3)\Big)\Big\}$$

We then publish this data to everyone who wants to participate in the generation of a zk-SNARK or its verification in the 3-factorization problem.

To understand how this common reference string can be used to evaluate polynomials at the secret evaluation point in the exponent of a generator, let's assume that we have deleted the simulation trapdoor. In that case, we have no way to know the secret evaluation point anymore, hence, we cannot evaluate polynomials at that point. However, we can evaluate polynomials of smaller degree than the degree of the target polynomial in the exponent of both generators at that point.

To see that, consider e.g. the polynomials $A_2(x) = 6x+10$ and $A_5(x) = 7x+4$ from the QAP of this problem. To evaluate these polynomials in the exponent of g_1 and g_2 at the secret point s without knowing the value of s (which is 2), we can use the common reference string and equation XXX. Using the scalar product notation instead of exponentiation, we get the following: add reference

$$\begin{aligned}
[A_2(s)]g_1 &= [6\cdot s^1 + 10\cdot s^0]g_1 \\
&= [6](33,34)+[10](13,15) && \# [s^0]g_1 = (13,15), [s^1]g_1 = (33,34) \\
&= [6\cdot 2](13,15)+[10](13,15) = [9](13,15) && \# \text{ logarithmic order on } \mathbb{G}_1 \\
&= (35,15) \\
[A_5(s)]g_1 &= [7\cdot s^1 + 4\cdot s^0]g_1 \\
&= [7](33,34)+[4](13,15) \\
&= [7\cdot 2](13,15)+[4](13,15) = [5](13,15) \\
&= (26,34)
\end{aligned}$$

Indeed, we are able to evaluate the polynomials in the exponent at a secret evaluation point, because that point is encrypted in the curve point $(33,34)$ and its secrecy is protected by the discrete logarithm assumption. Of course, in our computation, we recovered the secret point $s = 2$, but that was only possible because we have a group of logarithmic order in order to simplify our pen-and-paper computations. Such an order is infeasible for computing in cryptographically secure curves. We can do the same computation on $\mathbb{G}_2$ and get the following:

$$\begin{aligned}
[A_2(s)]g_2 &= [6\cdot s^1 + 10\cdot s^0]g_2 \\
&= [6](10v^2,28v^3)+[10](7v^2,16v^3) \\
&= [6\cdot 2](7v^2,16v^3)+[10](7v^2,16v^3) = [9](7v^2,16v^3) \\
&= (37v^2,16v^3) \\
[A_5(s)]g_2 &= [7\cdot s^1 + 4\cdot s^0]g_1 \\
&= [7](10v^2,28v^3)+[4](7v^2,16v^3) \\
&= [7\cdot 2](7v^2,16v^3)+[4](7v^2,16v^3) = [5](7v^2,16v^3) \\
&= (16v^2,28v^3)
\end{aligned}$$

Apart from the target polynomial T, all other polynomials of the quadratic arithmetic program can be evaluated in the exponent this way.

The Prover Phase Given some rank-1 constraint system R and instance $I = (I_1, \ldots, I_n)$, the task of the prover phase is to convince any verifier that a prover knows a witness W to instance I such that $(I;W)$ is a word in the language L_R of the system, without revealing anything about W.

To achieve this in the Groth_16 protocol, we assume that any prover has access to the rank-1 constraint system of the problem, in addition to some algorithm that tells the prover how to compute constructive proofs for the R1CS. In addition, the prover has access to a common reference string and its associated quadratic arithmetic program.

In order to generate a zk-SNARK for this instance, the prover first computes a valid constructive proof as explained in XXX, that is, the prover generates a proper witness $W = (W_1, \ldots, W_m)$ such that $(I_1, \ldots, I_n; W_1, \ldots, W_m)$ is a solution to the rank-1 constraint system R.

add reference

The prover then uses the quadratic arithmetic program and computes the polynomial $P_{(I;W)}$, as explained in 6.15. They then divide $P_{(I;W)}$ by the target polynomial T of the quadratic arithmetic program. Since $P_{(I;W)}$ is constructed from a valid solution to the R1CS, we know from 6.15 that it is divisible by T. This implies that polynomial division of P by T generates another polynomial $H := P/T$, with $deg(H) < deg(T)$.

check reference

check reference

The prover then evaluates the polynomial $(H \cdot T)\delta^{-1}$ in the exponent of the generator g_1 at the secret point s, as explained in XXX. To see how this can be achieved, let $H(x)$ be the quotient polynomial P/T:

add reference

$$H(x) = H_0 \cdot x^0 + H_1 \cdot x^1 + \ldots + H_k \cdot x^k \tag{8.3}$$

To evaluate $H \cdot T$ at s in the exponent of g_1, the prover uses the common reference string and computes as follows:

$$g_1^{\frac{H(s)\cdot T(s)}{\delta}} = \left(g_1^{\frac{s^0 \cdot T(s)}{\delta}}\right)^{H_0} \cdot \left(g_1^{\frac{s^1 \cdot T(s)}{\delta}}\right)^{H_1} \cdots \left(g_1^{\frac{s^k \cdot T(s)}{\delta}}\right)^{H_k}$$

After this has been done, the prover samples two random field elements $r, t \in \mathbb{F}_r$, and uses the common reference string, the instance variables $I_1, \ldots, I_n$ and the witness variables $W_1, \ldots, W_m$ to compute the following curve points:

$$\begin{aligned}
g_1^W &= \left(g_1^{\frac{\beta \cdot A_{1+n}(s) + \alpha \cdot B_{1+n}(s) + C_{1+n}(s)}{\delta}}\right)^{W_1} \cdots \left(g_1^{\frac{\beta \cdot A_{m+n}(s) + \alpha \cdot B_{m+n}(s) + C_{m+n}(s)}{\delta}}\right)^{W_m} \\
g_1^A &= g_1^\alpha \cdot g_1^{A_0(s)} \cdot \left(g_1^{A_1(s)}\right)^{I_1} \cdots \left(g_1^{A_n(s)}\right)^{I_n} \cdot \left(g_1^{A_{n+1}(s)}\right)^{W_1} \cdots \left(g_1^{A_{n+m}(s)}\right)^{W_m} \cdot \left(g_1^\delta\right)^r \\
g_1^B &= g_1^\beta \cdot g_1^{B_0(s)} \cdot \left(g_1^{B_1(s)}\right)^{I_1} \cdots \left(g_1^{B_n(s)}\right)^{I_n} \cdot \left(g_1^{B_{n+1}(s)}\right)^{W_1} \cdots \left(g_1^{B_{n+m}(s)}\right)^{W_m} \cdot \left(g_1^\delta\right)^t \\
g_2^B &= g_2^\beta \cdot g_2^{B_0(s)} \cdot \left(g_2^{B_1(s)}\right)^{I_1} \cdots \left(g_2^{B_n(s)}\right)^{I_n} \cdot \left(g_2^{B_{n+1}(s)}\right)^{W_1} \cdots \left(g_2^{B_{n+m}(s)}\right)^{W_m} \cdot \left(g_2^\delta\right)^t \\
g_1^C &= g_1^W \cdot g_1^{\frac{H(s)\cdot T(s)}{\delta}} \cdot \left(g_1^A\right)^t \cdot \left(g_1^B\right)^r \cdot \left(g_1^\delta\right)^{-r \cdot t}
\end{aligned}$$

In this computation, the group elements $g_1^{A_j(s)}$, $g_1^{B_j(s)}$ and $g_2^{B_j(s)}$ can be derived from the common reference string and the quadratic arithmetic program of the problem, as we have seen in XXX. In fact, those points only have to be computed once, and can be published and reused

add reference

for multiple proof generations because they are the same for all instances and witnesses. All other group elements are part of the common reference string.

After all these computations have been done, a valid zero-knowledge succinct non-interactive argument of knowledge π in the Groth_16 protocol is given by the following three curve points:

$$\pi = (g_1^A, g_1^C, g_2^B) \tag{8.4}$$

As we can see, a Groth_16 zk-SNARK consists of 3 curve points, two points from $\mathbb{G}_1$ and 1 point from $\mathbb{G}_2$. The argument is specifically designed this way because, in typical applications, $\mathbb{G}_1$ is a torsion group of an elliptic curve over some prime field, while $\mathbb{G}_2$ is a subgroup of a torsion group over an extension field. Elements from $\mathbb{G}_1$ therefore need less space to be stored, and computations in $\mathbb{G}_1$ are typically faster then in $\mathbb{G}_2$.

Since the witness is encoded in the exponent of a generator of a cryptographically secure elliptic curve, it is hidden from anyone but the prover. Moreover, since any proof is randomized by the occurrence of the random field elements r and t, proofs are not unique to any given witness. This is an important feature because, if all proofs for the same witness would be the same, knowledge of a witness would destroy the zero-knowledge property of those proofs.

Example 138 (The 3-factorization Problem). To see how a prover might compute a zk-SNARK, consider the 3-factorization problem from 106, our protocol parameters from XXX as well as the common reference string from XXX.

check reference

Our task is to compute a zk-SNARK for the instance $I_1 = 11$ and its constructive proof $(W_1, W_2, W_3, W_4) = (2,3,4,6)$ as computed in XXX. As we know from 6.15, the associated polynomial $P_{(I;W)}$ of the quadratic arithmetic program from XXX is given by the following equation:

add reference

add reference

$$P_{(I;W)} = x^2 + x + 9$$

Since $P_{(I;W)}$ is identical to the target polynomial $T(x) = x^2 + x + 9$ in this example, we know from XXX that the quotient polynomial $H = P/T$ is the constant degree 0 polynomial:

add reference

$$H(x) = H_0 \cdot x^0 = 1 \cdot x^0$$

check reference

We therefore use $[\frac{s^0 \cdot T(s)}{\delta}]g_1 = (26,34)$ from our common reference string XXX of the 3-factorization problem and compute as follows:

add reference

$$\begin{aligned}
[\frac{H(s)\cdot T(s)}{\delta}]g_1 &= [H_0](26,34) = [1](26,34) \\
&= (26,34)
\end{aligned}$$

add reference

add reference

In the next step, we have to compute all group elements required for a proper Groth16 zk-SNARK. We start with g_1^W. Using scalar products instead of the exponential notation, and $\oplus$ for the group law on the `BLS6_6` curve, we have to compute the point $[W]g_1$:

$$\begin{aligned}
[W]g_1 = \; & [W_1]g_1^{\frac{\beta\cdot A_2(s)+\alpha\cdot B_2(s)+C_2(s)}{\delta}} \oplus [W_2]g_1^{\frac{\beta\cdot A_3(s)+\alpha\cdot B_3(s)+C_3(s)}{\delta}} \oplus [W_3]g_1^{\frac{\beta\cdot A_4(s)+\alpha\cdot B_4(s)+C_4(s)}{\delta}} \\
& \oplus [W_4]g_1^{\frac{\beta\cdot A_5(s)+\alpha\cdot B_5(s)+C_5(s)}{\delta}}
\end{aligned}$$

To compute this point, we have to remember that a prover should not be in possession of the simulation trapdoor, hence, they do not know what α, β, δ and s are. In order to compute this group element, the prover therefore needs the common reference string. Using the logarithmic order from XXX and the witness, we get the following:

add reference

$$
\begin{aligned}
[W]g_1 &= [2](33,34) \oplus [3](26,34) \oplus [4](38,28) \oplus [6](35,28)\\
&= [2\cdot 2](13,15) \oplus [3\cdot 5](13,15) \oplus [4\cdot 10](13,15) \oplus [6\cdot 4](13,15)\\
&= [2\cdot 2+3\cdot 5+4\cdot 10+6\cdot 4](13,15) = [5](13,15)\\
&= (26,34)
\end{aligned}
$$

In a next step, we compute g_1^A. We sample the random point $r = 11$ from $\mathbb{F}_{13}$, using scalar products instead of the exponential notation, and $\oplus$ for the group law on the `BLS6_6` curve. We then have to compute the following expression:

$$
\begin{aligned}
[A]g_1 = \; & [\alpha]g_1 \oplus [A_0(s)]g_1 \oplus [I_1][A_1(s)]g_1 \oplus [W_1][A_2(s)]g_1 \oplus [W_2][A_3(s)]g_1\\
& \oplus [W_3][A_4(s)]g_1 \oplus [W_4][A_5(s)]g_1 \oplus [r][\delta]g_1
\end{aligned}
$$

Since we don't know what α, δ and s are, we look up $[\alpha]g_1$ and $[\delta]g_1$ from the common reference string. Recall from XXX that we can evaluate $[A_j(s)]g_1$ without knowling the secret evaluation point s. According to XXX, we have $[A_2(s)]g_1 = (35,15)$, $[A_5(s)]g_1 = (26,34)$ and $[A_j(s)]g_1 = \mathcal{O}$ for all other indices $0 \le j \le 5$. Since $\mathcal{O}$ is the neutral element on $\mathbb{G}_1$, we get the following:

add reference

add reference

$$
\begin{aligned}
[A]g_1 &= (27,34) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2](35,15) \oplus [3]\mathcal{O} \oplus [4]\mathcal{O} \oplus [6](26,34) \oplus [11](38,15)\\
&= (27,34) \oplus [2](35,15) \oplus [6](26,34) \oplus [11](38,15)\\
&= [6](13,15) \oplus [2\cdot 9](13,15) \oplus [6\cdot 5](13,15) \oplus [11\cdot 3](13,15)\\
&= [6+2\cdot 9+6\cdot 5+11\cdot 3](13,15) = [9](13,15)\\
&= (35,15)
\end{aligned}
$$

In order to compute the two curve points $[B]g_1$ and $[B]g_2$, we sample another random element $t = 4$ from $\mathbb{F}_{13}$. Using the scalar product instead of the exponential notation, and $\oplus$ for the group law on the `BLS6_6` curve, we have to compute the following expressions:

$$
\begin{aligned}
[B]g_1 = \; & [\beta]g_1 \oplus [B_0(s)]g_1 \oplus [I_1][B_1(s)]g_1 \oplus [W_1][B_2(s)]g_1 \oplus [W_2][B_3(s)]g_1\\
& \oplus [W_3][B_4(s)]g_1 \oplus [W_4][B_5(s)]g_1 \oplus [t][\delta]g_1\\
[B]g_2 = \; & [\beta]g_2 \oplus [B_0(s)]g_2 \oplus [I_1][B_1(s)]g_2 \oplus [W_1][B_2(s)]g_2 \oplus [W_2][B_3(s)]g_2\\
& \oplus [W_3][B_4(s)]g_2 \oplus [W_4][B_5(s)]g_2 \oplus [t][\delta]g_2
\end{aligned}
$$

Since we don't know what β, δ and s are, we look up the associated group elements from the common reference string. Recall from XXX that we can evaluate $[B_j(s)]g_1$ without knowing the secret evaluation point s. Since $B_3 = A_2$ and $B_4 = A_5$, we have $[B_3(s)]g_1 = (35,15)$, $[B_4(s)]g_1 = (26,34)$ according to XXX, and $[B_j(s)]g_1 = \mathcal{O}$ for all other indices $0 \le j \le 5$. Since $\mathcal{O}$ is the neutral element on $\mathbb{G}_1$, we get the following:

add reference

add reference

$$
\begin{aligned}
[B]g_1 &= (26,34) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2]\mathcal{O} \oplus [3](35,15) \oplus [4](26,34) \oplus [6]\mathcal{O} \oplus [4](38,15)\\
&= (26,34) \oplus [3](35,15) \oplus [4](26,34) \oplus [4](38,15)\\
&= [5](13,15) \oplus [3\cdot 9](13,15) \oplus [4\cdot 5](13,15) \oplus [4\cdot 3](13,15)\\
&= [5+3\cdot 9+4\cdot 5+4\cdot 3](13,15) = [12](13,15)\\
&= (13,28)
\end{aligned}
$$

$$
\begin{aligned}
[B]g_2 &= (16v^2, 28v^3) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2]\mathcal{O} \oplus [3](37v^2, 16v^3) \oplus [4](16v^2, 28v^3) \oplus [6]\mathcal{O} \oplus [4](42v^2, 16v^3) \\
&= (16v^2, 28v^3) \oplus [3](37v^2, 16v^3) \oplus [4](16v^2, 28v^3) \oplus [4](42v^2, 16v^3) \\
&= [5](7v^2, 16v^3) \oplus [3 \cdot 9](7v^2, 16v^3) \oplus [4 \cdot 5](7v^2, 16v^3) \oplus [4 \cdot 3](7v^2, 16v^3) \\
&= [5 + 3 \cdot 9 + 4 \cdot 5 + 4 \cdot 3](7v^2, 16v^3) = [12](7v^2 + 16v^3) \\
&= (7v^2, 27v^3)
\end{aligned}
$$

In a last step, we combine the previous computations to compute the point $[C]g_1$ in the group $\mathbb{G}_1$ as follows:

$$
\begin{aligned}
[C]g_1 &= [W]g_1 \oplus [\frac{H(s) \cdot T(s)}{\delta}]g_1 \oplus [t][A]g_1 \oplus [r][B]g_1 \oplus [-r \cdot t][\delta]g_1 \\
&= (26, 34) \oplus (26, 34) \oplus [4](35, 15) \oplus [11](13, 28) \oplus [-11 \cdot 4](38, 15) \\
&= [5](13, 15) \oplus [5](13, 15) \oplus [4 \cdot 9](13, 15) \oplus [11 \cdot 12](13, 15) \oplus [-11 \cdot 4 \cdot 3](13, 15) \\
&= [5 + 5 + 4 \cdot 9 + 11 \cdot 12 - 11 \cdot 4 \cdot 3](13, 15) = [7](13, 15) \\
&= (27, 9)
\end{aligned}
$$

Given the instance $I_1 = 11$, we can now combine these computations and see that the following 3 curve points are a zk-SNARK for the witness $(W_1, W_2, W_3, W_4) = (2, 3, 4, 6)$:

$$
\pi = ((35, 15), (27, 9), (7v^2, 27v^3))
$$

We ca now publish this zk-SNARK, or send it to a designated verifier. Note that, if we had sampled different values for r and t, we would have computed a different SNARK for the same witness. The SNARK, therefore, hides the witness perfectly, which means that it is impossible to reconstruct the witness from the SNARK.

The Verification Phase Given some rank-1 constraint system R, instance $I = (I_1, \ldots, I_n)$ and zk-SNARK π, the task of the verification phase is to check that π is indeed an argument for a constructive proof. Assuming that the simulation trapdoor does not exists anymore and the verification checks the proof, the verifier can be convinced that someone knows a witness $W = (W_1, \ldots, W_m)$ such that $(I; W)$ is a word in the language of R.

To achieve this in the Groth16 protocol, we assume that any verifier is able to compute the pairing map $e(\cdot, \cdot)$ efficiently, and has access to the common reference string used to produce the SNARK π. In order to verify the SNARK with respect to the instance $(I_1, \ldots, I_n)$, the verifier computes the following curve point:

$$
g_1^I = \left(g_1^{\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma}}\right) \cdot \left(g_1^{\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma}}\right)^{I_1} \cdots \left(g_1^{\frac{\beta \cdot A_n(s) + \alpha \cdot B_n(s) + C_n(s)}{\gamma}}\right)^{I_n}
$$

With this group element, the verifier is able to verify the SNARK $\pi = (g_1^A, g_1^C, g_2^B)$ by checking the following equation using the pairing map:

$$
e(g_1^A, e_2^B) = e(g_1^\alpha, g_2^\beta) \cdot e(g_1^I, g_2^\gamma) \cdot e(g_1^C, g_2^\delta) \tag{8.5}
$$

If the equation holds true, the SNARK is accepted. If the equation does not hold, the SNARK is rejected.

Remark 6. We know from chapter 5 that computing pairings in cryptographically secure pairing groups is computationally expensive. As we can see, in the Groth16 protocol, 3 pairings are required to verify the SNARK, because the pairing $e(g_1^{\alpha}, g_2^{\beta})$ is independent of the proof, meaning that can be computed once and then stored as an amendment to the verifier key.

check reference

In **?**, the author showed that 2 is the minimal amount of pairings that any protocol with similar properties has to use. This protocol is therefore close to the theoretical minimum. In the same paper, the author outlined an adaptation that only uses 2 pairings. However, that reduction comes with the price of much more overhead computation. Having 3 pairings is therefore a compromise that gives the overall best performance. To date, the Groth16 protocol is the most efficient in its class.

Example 139 (The 3-factorization Problem). To see how a verifier might check a zk-SNARK for some given instance I, consider the 3-factorization problem from 106, our protocol parameters from XXX, the common reference string from XXX as well as the zk-SNARK $\pi = ((35,15),(27,9),(7v^2,27v^3))$, which claims to be an argument of knowledge for a witness for the instance $I_1 = 11$.

check reference

add reference

add reference

In order to verify the zk-SNARK for that instance, we first compute the curve point g_1^I. Using scalar products instead of the exponential notation, and $\oplus$ for the group law on the `BLS6_6` curve, we have to compute the point $[I]g_1$ as follows:

$$[I]g_1 = [\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma}]g_1 \oplus [I_1][\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma}]g_1$$

To compute this point, we have to remember that a verifier should not be in possession of the simulation trapdoor, which means that they do not know what α, β, γ and s are. In order to compute this group element, the verifier therefore needs the common reference string. Using the logarithmic order from XXX and instance I_1, we get the following:

add reference

$$\begin{aligned}
[I]g_1 &= [\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma}]g_1 \oplus [I_1][\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma}]g_1 \\
&= \mathcal{O} \oplus [11](33,9) \\
&= [11 \cdot 11](13,15) = [4](13,15) \\
&= (35,28)
\end{aligned}$$

In the next step, we have to compute all the pairings involved in equation XXX. Using the logarithmic order on $\mathbb{G}_1$ and $\mathbb{G}_2$ as well as the bilinearity property of the pairing map we get the following:

add reference

$$
\begin{aligned}
e([A]g_1,[B]g_2) &= e((35,15),(7v^2,27v^3)) = e([9](13,15),[12](7v^2,16v^3)) \\
&= e((13,15),(7v^2,16v^3))^{9\cdot 12} \\
&= e((13,15),(7v^2,16v^3))^{108} \\
e([\alpha]g_1,[\beta]g_2) &= e((27,34),(16v^2,28v^3)) = e([6](13,15),[5](7v^2,16v^3)) \\
&= e((13,15),(7v^2,16v^3))^{6\cdot 5} \\
&= e((13,15),(7v^2,16v^3))^{30} \\
e([I]g_1,[\gamma]g_2) &= e((35,28),(37v^2,27v^3)) = e([4](13,15),[4](7v^2,16v^3)) \\
&= e((13,15),(7v^2,16v^3))^{4\cdot 4} \\
&= e((13,15),(7v^2,16v^3))^{16} \\
e([C]g_1,[\delta]g_2) &= e((27,9),(42v^2,16v^3)) = e([7](13,15),[3](7v^2,16v^3)) \\
&= e((13,15),(7v^2,16v^3))^{7\cdot 3} \\
&= e((13,15),(7v^2,16v^3))^{21}
\end{aligned}
$$

In order to check equation XXX, observe that the target group $\mathbb{G}_T$ of the Weil pairing is a finite cyclic group of order 13. Exponentiation is therefore done in modular 13 arithmetics. Accordingly, since 108 mod $13 = 4$, we evaluate the left side of equation XXX as follows:

add reference

add reference

$$e([A]g_1,[B]g_2) = e((13,15),(7v^2,16v^3))^{108} = e((13,15),(7v^2,16v^3))^4$$

Similarly, we evaluate the right side of equation XXX using modular 13 arithmetics and the exponential law $a^x \cdot a^y = a^{x+y}$:

add reference

$$
\begin{aligned}
e([\alpha]g_1,[\beta]g_2)\cdot e([I]g_1,[\gamma]g_2)\cdot e([C]g_1,[\delta]g_2) &= \\
e((13,15),(7v^2,16v^3))^{30}\cdot e((13,15),(7v^2,16v^3))^{16}\cdot e((13,15),(7v^2,16v^3))^{21} &= \\
e((13,15),(7v^2,16v^3))^{4}\cdot e((13,15),(7v^2,16v^3))^{3}\cdot e((13,15),(7v^2,16v^3))^{8} &= \\
e((13,15),(7v^2,16v^3))^{4+3+8} &= \\
e((13,15),(7v^2,16v^3))^{2}
\end{aligned}
$$

As we can see, both the left and the right side of equation XXX are identical, which implies that the verification process accepts the simulated proof.

add reference

NOTE: UNFORTUNATELY NOT! :-((HENCE THERE IS AN ERROR SOMEWHERE ... NEED TO FIX IT AFTER VACATION

fix error

Proof Simulation During the execution of a setup phase, a common reference string is generated, along with a simulation trapdoor, the latter of which must be deleted at the end of the setup-phase. As an alternative, a more complicated multi-party protocol like [XXX] can be used to split the knowledge of the simulation trapdoor among many different parties.

add reference

In this paragraph, we will show why knowledge of the simulation trapdoor is problematic, and how it can be used to generate zk-SNARKs for a given instance without any knowledge or the existence of associated witness.

To be more precise, let I be an instance for some R1CS language L_R. We call a zk-SNARK for L_R **forged** or **simulated** if it passes a verification but its generation does not require the existence of a witness W such that $(I;W)$ is a word in L_R.

To see how simulated zk-SNARKs can be computed, assume that a forger has knowledge of proper Groth_16 parameters, a quadratic arithmetic program of the problem, a common reference string and its associated simulation trapdoor τ:

$$\tau = (\alpha, \beta, \gamma, \delta, s) \tag{8.6}$$

Given some instance I, the forger's task is to generate a zk-SNARK for this instance that passes the verification process, without having access to any other zk-SNARKs for this instance and without knowledge of a valid witness W.

To achieve this in the Groth_16 protocol, the forger can use the simulation trapdoor in combination with the QAP and two arbitrary field elements A and B from the scalar field $\mathbb{F}_r$ of the pairing groups to g_1^C compute for the instance $(I_1, \ldots, I_n)$ as follows:

$$g_1^C = g_1^{\frac{A \cdot B}{\delta}} \cdot g_1^{-\frac{\alpha \cdot \beta}{\delta}} \cdot g_1^{-\frac{\beta A_0(s)+\alpha B_0(s)+C_0(s)}{\delta}} \cdot \left(g_1^{-\frac{\beta A_1(s)+\alpha B_1(s)+C_1(s)}{\delta}}\right)^{I_1} \cdots \left(g_1^{-\frac{\beta A_n(s)+\alpha B_n(s)+C_n(s)}{\delta}}\right)^{I_n}$$

The forger then publishes the zk-SNARK $\pi_{forged} = (g_1^A, g_1^C, g_2^B)$, which will pass the verification process and is computable without the existence of a witness $(W_1, \ldots, W_m)$.

To see that the simulation trapdoor is necessary and sufficient to compute the simulated proof π_{forged}, first observe that both generators g_1 and g_2 are known to the forger, as they are part of the common reference string, encoded as $g_1^{s^0}$ and $g_2^{s^0}$. The forger is therefore able to compute $g_1^{A \cdot B}$. Moreover, since the forger knows α, β, δ and s from the trapdoor, they are able to compute all factors in the computation of g_1^C.

If, on the other hand, the simulation trapdoor is unknown, it is not possible to compute g_1^C, since, for example, the computational Diffie-Hellman assumption makes the derivation of $g_1^{\alpha \cdot \beta}$ from g_1^α and g_1^β infeasible.

Example 140 (The 3-factorization Problem). To see how a forger might simulate a zk-SNARK for some given instance I, consider the 3-factorization problem from 106, our protocol parameters from XXX, the common reference string from XXX and the simulation trapdoor $\tau = (6, 5, 4, 3, 2)$ of that CRS.

In order to forge a zk-SNARK for instance $I_1 = 11$, we don't need a constructive proof for the associated rank-1 constraint system, which implies that we don't have to execute the circuit $C_{3.fac}(\mathbb{F}_{13})$. Instead, we have to choose 2 arbitrary elements A and B from $\mathbb{F}_{13}$, and compute g_1^A, g_2^B and g_1^C as defined in XXX. We choose $A = 9$ and $B = 3$, and, since $\delta^{-1} = 3$, we compute as follows:

check reference

add reference

add reference

add reference

$$\begin{aligned}
[A]g_1 =& [9](13,15) = (35,15)\\
[B]g_2 =& [3](7v^2, 16v^3) = (42v^2, 16v^3)\\
[C]g_1 =& [\frac{A \cdot B}{\delta}]g_1 \oplus [-\frac{\alpha \cdot \beta}{\delta}]g_1 \oplus [-\frac{\beta A_0(s) + \alpha B_0(s) + C_0(s)}{\delta}]g_1 \oplus\\
& [I_1][-\frac{\beta A_1(s) + \alpha B_1(s) + C_1(s)}{\delta}]g_1\\
=& [(9 \cdot 3) \cdot 9](13,15) \oplus [-(6 \cdot 5) \cdot 9](13,15) \oplus [0](13,15) \oplus [11][-(7 \cdot 2 + 4) \cdot 9](13,15)\\
=& [9](13,15) \oplus [3](13,15) \oplus [12](13,15) = [11](13,15)\\
=& (33,9)
\end{aligned}$$

This is all we need to generate our forged proof for the 3-factorization problem. We publish the simulated zk-SNARK:

$$\pi_{fake} = ((35,15),(33,9),(42v^2,16v^3))$$

Despite the fact that this zk-SNARK was generated without knowledge of a proper witness, it is indistinguishable from a zk-SNARK that proves knowledge of a proper witness.

To see that, we show that our forged SNARK passes the verification process. In order to verify π_{fake}, we proceed as in XXX, and compute the curve point g_1^I for the instance $I_1 = 11$. Since the instance is the same as in example XXX, we can parallel the computation from XXX:

add reference

add reference

add reference

$$\begin{aligned}
[I]g_1 &= [\frac{\beta\cdot A_0(s)+\alpha\cdot B_0(s)+C_0(s)}{\gamma}]g_1 \oplus [I_1][\frac{\beta\cdot A_1(s)+\alpha\cdot B_1(s)+C_1(s)}{\gamma}]g_1\\
&= (35,28)
\end{aligned}$$

In a next step we have to compute all the pairings involved in equation XXX. Using the logarithmic order on $\mathbb{G}_1$ and $\mathbb{G}_2$ as well as the bilinearity property of the pairing map we get

add reference

$$\begin{aligned}
e([A]g_1,[B]g_2) &= e((35,15),(42v^2,16v^3)) = e([9](13,15),[3](7v^2,16v^3))\\
&= e((13,15),(7v^2,16v^3))^{9\cdot 3}\\
&= e((13,15),(7v^2,16v^3))^{27}\\
e([\alpha]g_1,[\beta]g_2) &= e((27,34),(16v^2,28v^3)) = e([6](13,15),[5](7v^2,16v^3))\\
&= e((13,15),(7v^2,16v^3))^{6\cdot 5}\\
&= e((13,15),(7v^2,16v^3))^{30}\\
e([I]g_1,[\gamma]g_2) &= e((35,28),(37v^2,27v^3)) = e([4](13,15),[4](7v^2,16v^3))\\
&= e((13,15),(7v^2,16v^3))^{4\cdot 4}\\
&= e((13,15),(7v^2,16v^3))^{16}\\
e([C]g_1,[\delta]g_2) &= e((33,9),(42v^2,16v^3)) = e([11](13,15),[3](7v^2,16v^3))\\
&= e((13,15),(7v^2,16v^3))^{11\cdot 3}\\
&= e((13,15),(7v^2,16v^3))^{33}
\end{aligned}$$

In order to check equation XXX, observe that the target group $\mathbb{G}_T$ of the Weil pairing is a finite cyclic group of order 13. Exponentiation is therefore done in modular 13 arithmetics. Using this, we evaluate the left side of equation XXX as follows:

add reference

add reference

$$e([A]g_1,[B]g_2) = e((13,15),(7v^2,16v^3))^{27} = e((13,15),(7v^2,16v^3))^1$$

since 27 mod 13 = 1. Similarly, we evaluate the right side of equation XXX using modular 13 arithmetics and the exponential law $a^x \cdot a^y = a^{x+y}$. We get

add reference

$$\begin{aligned}
e([\alpha]g_1,[\beta]g_2)\cdot e([I]g_1,[\gamma]g_2)\cdot e([C]g_1,[\delta]g_2) &=\\
e((13,15),(7v^2,16v^3))^{30}\cdot e((13,15),(7v^2,16v^3))^{16}\cdot e((13,15),(7v^2,16v^3))^{33} &=\\
e((13,15),(7v^2,16v^3))^{4}\cdot e((13,15),(7v^2,16v^3))^{3}\cdot e((13,15),(7v^2,16v^3))^{7} &=\\
e((13,15),(7v^2,16v^3))^{4+3+7} &=\\
e((13,15),(7v^2,16v^3))^{1}
\end{aligned}$$

As we can see, both the left and the right side of equation XXX are identical, which implies that the verification process accepts the simulated proof. π_{fake} therefore convinces the verifier that a witness to the 3-factorization problem exists. However, no such witness was really necessary to generate the proof.

add reference

Chapter 9

Exercises and Solutions

TODO: All exercises we provided should have a solution, which we give here in all detail.