Moonmath manual

June 16, 2021

Lorem **ipsum** dolor sit amet, consectetur adipiscing elit. Pellentesque semper viverra dictum. Fusce interdum venenatis leo varius vehicula. Etiam ac massa dolor. Quisque vel massa faucibus, facilisis nulla nec, egestas lectus. Sed orci dui, egestas non felis vel, fringilla pretium odio. *Aliquam* vel consectetur felis. Suspendisse justo massa, maximus eget nisi a, maximus gravida mi.

Here is a citation for demonstration: **?**

Chapter 1

Introduction

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

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 academiccommunity 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 community 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.

Chapter 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 3

Preliminaries

Introduction and summary of what we do in this chapter

3.1 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

3.2 SNARKS

3.3 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.

3.3.1 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.

3.4 Software Used in This Book

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 4

Number Theory

To understand the internals SNARKs it is foremost important to understand the basics of finite field arithmetics AND STUFF.

We therefore start with a brief introduction to fundamental algebraic terms like fields, field extensions AND STUFF . 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, that likely all of us know and do basic computations with these examples.

The motivated reader is then encuraged to do some exercises from the apendix of this chapter.

4.1 Preliminaries

INTO-BLA

When you read papers about cryptography or mathematical papers in general, you will frequently stumble across terms like *fields*, *groups*,*rings* and similar. As we will use these terms repeatedly, we have to start this chapter with a short introduction to those terms.

In a nutshell, those terms define sets that are in some sense analog to numbers, in that you can add, subtract, multiply or divide. Or more abstractly those sets have certain ways to combine two elements into a new element.

We know many example like the natural numbers, the integers, the ratinal or the real numbers from school and they are in some sense already the most fundamental examples

Lets start with the concept of a group. Remember back in school when we learned about addition and subtraction of integers. We learned that we can always add to integers and that the result is an integer again. We also learned that nothing happens, when we add zero to any integer, that it doesn't matter in which order we add a given set of integers and that for every integer there is always a negative counterpart and if we add both together we get zero.

Distilling these rules to the smallest list of properties and make rthem abstract we arrive at the definition of a group:

Definition 4.1.0.1 (Group)**.** *A grou $(\mathbb{G},\cdot)$ is a set $\mathbb{G}$, together with a map $\cdot : \mathbb{G}\cdot\mathbb{G}\to\mathbb{G}$, called the group law (or addition respectively multiplication), such that the following 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.*

In addition a group is called commutative if the equation $g_1 \cdot g_2 = g_2 \cdot g_1$ holds for all $g_1, g_2 \in \mathbb{G}$.

A group is called finite, if the underlying set of elements is finite.

*An element $g \in G$ is called a **generator** of that group if every other group element can be generated by combining g with itself only.*

Remark 1. *By definition, a generator is a special element, that can generate the entire group, by adding it repeatedly to itself. For example the number* 1 *is a generator for the natural numbers (not a group), as you can compute any natural number by adding 1 repeatedly to itself.*

Remark 2. *If not stated otherwise, we will use the* $+$ *symbol for the group law of a commutative group. In that case we use the symbol* 0 *for the neutral element and write the inverse as* $-x$.

Sagemath comes with a definition of the category of all groups, which inherits the properties of *Magmas*, which are like groups but without the notion of inverses. For our purposes we only need commutative groups and in particular finite and cyclic groups.

```
sage: Groups()
Category of groups
sage: CommutativeAdditiveGroups()
Category of commutative additive groups
sage: FiniteGroups()
Category of finite groups
```

Example 1 (Integer Addition and Subtraction). *As previously described, the set of integers together with integer addition is the archetypical example of a group. In contrast to our definition above the group laws is usually not written as* $\cdot$, *but as* $+$ *instead. The neutral element is the number* 0 *and the inverse of a natural number* $x \in \mathbb{Z}$ *is the negative* $-x$. *Since* $x + y = y + x$ *this is an example of a commutative group.*

Example 2 (The trivial group). *The most basic example of a group, is group with just one element* $\{\bullet\}$ *and the group law* $0 \cdot 0 = 0$. *This group can be defined in many ways. In sage it is defined as the group of all permutations of a single element (which is just the identity permutation that is doing nothing). Therefor in sage we can write*

```
sage: TrivialGroup = SymmetricGroup(1)
```

Now thinking of integers again, we know, that there are actually two operations addition and multiplication, such that that they interact in a certain way. This is captured by the concept of a ring, where integers are the most basic example in a sense

Definition 4.1.0.2 (Commutative Ring with Unit). *A ring* $(R, +, \cdot)$ *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:*

- *$(R, +)$ is a commutative group, where the neutral element is denoted here with 0.*
- *(Existence of a unit) There is an element $1 \in R$ 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 apply:*

$$g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3 \quad \text{and} \quad (g_1 + g_2) \cdot g_3 = g_1 \cdot g_3 + g_2 \cdot g_3$$

- *COMMUTATIVITY*

```
sage: CommutativeRings()
Category of commutative rings
sage: CommutativeRings().super_categories()
[Category of rings, Category of commutative monoids]
```

Example 3 (The Ring of Integers)**.** *The set $\mathbb{Z}$ of integers with the usual addition and multiplication is a ring. In sage the ring of integers is called as follows and as we know, not all elements have multiplicative inverses*

```
sage: ZZ # A sage notation for the Ring of integers
Integer Ring
sage: ZZ(5) # Get an element from the Ring of integers
5
sage: ZZ(5) + ZZ(3)
8
sage: ZZ(5) * ZZ(3)
15
sage: ZZ.random_element(10**50)
33757040729174880344438969757679342512630006686532
sage: ZZ(27713).str(2) # Binary string representation
110110001000001
sage: ZZ(27713).str(16) # Hexadecimal string representation
6c41
```

Example 4 (Underlying additive group of a ring)**.** *Every ring $(R, +, \cdot)$ gives rise to group, if we just forget about the multiplication*

The following example is more interesting, as it introduces the concept of so called addition and multiplication table. To understand those tables BLA....

Example 5. *Let $S := \{\bullet, \star, \odot, \otimes\}$ be a set that contains four elements and let adiition 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 an "alien equation" can be solved, we have to keep in mind, that rings behaves mostly like normal number when it comes to bracketing and computation rules. The only differences are the symbols and the actual way to add and multiply. With this we solve the equation for x in the "usual way"

$$\begin{aligned}
\otimes \circ (x \cup \odot) &= \star && \textit{distributive law}\\
\otimes \circ x \cup \otimes \circ \odot &= \star && \otimes \circ \odot = \odot\\
\otimes \circ x \cup \odot &= \star && \textit{add the additive inverse to both sides}\\
\otimes \circ x \cup \odot \cup -\odot &= \star \cup -\odot &&\\
\otimes \circ x \cup \bullet &= \star \cup -\odot && \textit{inverse property}\\
\otimes \circ x &= \star \cup -\odot && \textit{neutral element can be deleted}\\
\otimes \circ x &= \star \cup \odot && -\odot = \odot \textit{from addition table}\\
\otimes \circ x &= \otimes && \star \cup \odot = \otimes \textit{from addition table}\\
(\otimes)^{-1} \circ \otimes \circ x &= (\otimes)^{-1} \circ \otimes && \textit{multiply with the multiplicative inverse}\\
\otimes \circ \otimes \circ x &= \otimes \circ \otimes &&\\
\star \circ x &= \star &&\\
x &= \star &&
\end{aligned}$$

On the other hand, EQUATION $2x = 3$ has no sultion and equation BLA has more then one solution. So rings are sometimes different from numbers. When we want more we need fields .... BLAHHHH EXPAND

It is a good and fundamental exercise in mathematics to understand examples like this in detail as it helps the reader to loosen the connection to what "should be" numbers and what not.

Definition 4.1.0.3 (Field)**.** *A field $(\mathbb{F}, +, \cdot)$ is a set $\mathbb{F}$, provided 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 holds*

- $(\mathbb{F}, +)$ *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 laws apply:*

$$g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3 \quad \textit{and} \quad (g_1 + g_2) \cdot g_3 = g_1 \cdot g_3 + g_2 \cdot g_3$$

The characteristic of a field $\mathbb{F}$ 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 a $n > 0$ exists, the field is also called of finite characteristic. If, on the other hand, every finite sum of 1 is not equal to zero, then the field is defined to have characteristic 0.

So a field is a commutative ring with unit, such that every element other the the neutral element of addition has an inverse.

```
sage: Fields()
Category of fields
```

Example 6 (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. In sage rational numbers are called like this*

```
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 7 (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 we always have* $0 \neq 1$ *in a field. From this follows that the smallest field must contain at least two elements and 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

For reasons we will understand better in XXX, sage defines this field as a so called Galois field with 2 elements. It is called like this:

```
sage: GF(2)
Finite Field of size 2
sage: GF(2)(1) # Get an element from GF(2)
1
sage: GF(2)(1) + GF(2)(1) # Addition
0
sage: GF(2)(1) / GF(2)(1) # Division
1
```

4.1.1 Integer Arithmetics

We start by a recapitulation of basic arithmetics as most of us will probably recall from school.

As we have seen in the previous section integers form a ring and division is not well defined. However one of the most basic but at the same time highly important technique, that every reader must become familiar with is the following (kind of) replacement for devision of integers called *Euklidean division* (**?**

Theorem 4.1.1.1 (Euklidean Division). *Let $a \in \mathbb{Z}$ be an integer and $b \in \mathbb{N}$ a natural number. Then there are always two numbers $m \in \mathbb{Z}$ and $r \in \mathbb{N}$, with $0 \leq r < b$ such that*

$$a = m \cdot b + r \tag{4.1}$$

This decomposition of a given b is called ***Euklidean division*** *or* ***division with remainder*** *and a is called the* ***divident****, b is called the* ***divisor****,m is called the quotient and r is called the* ***remainder****.*

Remark 3. *If a, b, m and r satisfy the equation (4.1), we often write a div $b := m$ to describe the quotient and use the symbol a mod $b := r$ for the remainder. We also say, that an integer a is divided by a number b if a mod $b = 0$ holds. In this case we also write $a|b$ (? Note 1 below).*

Example 8. *Using our previously defined notation, we have -17 div $4 = -5$ and -17 mod $4 = 3$, because $-17 = -5 \cdot 4 + 3$ is the Euklidean division of -17 and 4 (Since the remainder is by definition a non-negative number). In this case 4 does not divide -17 as the reminder is not zero. Writing $-17|4$ therefore has no meaning.*

On the other hand we can write $12|4$, since 4 divides 12, as 12 mod $4 = 0$.

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

A fundamental property of Euclidean division is that both the quotient and the remainder exist and are unique. The result is therefore independent of any algorithm that actually does the computation.

Methods for computing the division with remainder are called *integer division algorithms*. Probably the best known algorithm is the so called *long division*, that most of us might have learned in school. It should be noted however that there are faster algorithms like *Newton–Raphson division* known.

As long division is the standard algorithm used for pen-and-paper division of multi-digit numbers expressed in decimal notation, the reader should become familiar with this algorithm as we use it all over this book when we do simple pen-and-paper computations.

In a nutshell, the algorithm shifts gradually from the left to the right end of the dividend, 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 final difference is then the remainder. To be more precise one version of the algorithm looks like this:

```
Divide $n$ by $d$
If $d = 0$ then error(DivisionByZeroException) end
$Q \leftarrow 0$ – Initialize quotient to zero
$R \leftarrow 0$ – Initialize remainder to zero
TO APPEAR
```

Example 9 (Integer Log Division). *To give an example of the basic integer long division algorithm most of us learned at school, lets divide the integer* 157843853 *by the number* 261*.*

So the goal is to find quotient and reminder $m, r \in \mathbb{N}$ such that

$$157843853 = 261 * m + r$$

holds. Using a notation that is mostly used in Commonwealth countries we can then compute

TO-APPEAR

```
sage: ZZ(157843853).quo_rem(ZZ(261)) # Euclidean Division
(604765, 188)
sage: ZZ(604765)*ZZ(261) + ZZ(188) # check
157843853
```

Another important algorithm frequently used in computations with integers is the so-called *extended Euclidean algorithm*, which calculates the greatest common divisor $gcd(a, b)$ of two natural numbers a and $b \in \mathbb{N}$, as well as two additional integers $s, t \in \mathbb{Z}$, such that the equation

$$gcd(a, b) = s \cdot a + t \cdot b \qquad (4.2)$$

holds. Two numbers are called **relative prime**, if their greates common divisor is 1.

The following pseudocode shows in detail how to calculate these numbers with the extended Euclidean algorithm (**?** chapter 2.9):

Definition 4.1.1.2. *Let the natural numbers $a, b \in \mathbb{N}$ be given. Then the so-called extended Euclidean algorithm is given by the following calculation rule:*

```
$r_0 := a, \quad r_1 := b, \quad s_0 := 1, \quad s_1 := 0, \quad k := 1$
while $r_k \neq 0$ do
    $q_k := r_{k-1}$ div $r_k$
    $r_{k+1} := r_{k-1} - q_k \cdot r_k$
    $s_{k+1} := s_{k-1} - q_k \cdot s_k$
    $k \leftarrow k + 1$
end while
```

As a result, the algorithm computes the integers $gcd(a, b) := r_k$, as well as $s := s_k$ and $t := (r_k - s_k \cdot a)$ div b such that the equation $gcd(a, b) = s \cdot a + t \cdot b$ holds.

Example 10. *To illustrate the algorithm, lets apply it to the numbers* $(12, 5)$*. We compute*

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 one can see that 12 *and* 5 *are relatively prime (since their greatest common divisor is $gcd(12, 5) = 1$) and that the equation $1 = (-2) \cdot 12 + 5 \cdot 5$ holds.*

```
sage: ZZ(12).xgcd(ZZ(5)) # (gcd,s,t)
(1, -2, 5)
```

4.1.2 Modular arithmetic

Congruence or modlar arithmetics (sometimes also called residue class arithmetics) is of central importance for understanding most modern crypto systems. In this section we will therefore take a closer look at this arithmetic. For the notation in cryptology see also **?** Chapter 3, or **?** Chapter 3.

MORE-HIGH-LEVEL-DESCRIPTION

Utilizing Euklidean division as explained previously (4.1.1), congruency of two integers with respect to a so-called moduli can be defined as follows (**?** chapter 3.1):

Definition 4.1.2.1 (congruency). *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 equation*

$$a \ mod \ n = b \ mod \ n \tag{4.3}$$

holds. In this case we write $a \equiv b \pmod n$. If two numbers are not congruent with respect to a given modulus n, we call them incongruent w.r.t. n.

So in other words, if some modulus n is given, then two integers are congruent with respect to this modulus if both Euclidean divisions by n give the same remainder.

Example 11. *To give a simple example, lets assume that we choose the modulus* 271. *Then we have*

$$7 \equiv 2446 \pmod{271}$$

since both $7 \ mod \ 271 = 7$ *as well as* $2446 \ mod \ 271 = 7$

```
sage: ZZ(7) % ZZ(271) == ZZ(2446) % ZZ(271)
True
```

The following theorem describes a fundamental property of modulus arithmetic, which is not known in the traditional arithmetics of integers: (**?** chapter 3.11):

Theorem 4.1.2.2 (Fermat's Little Theorem). *Let $p \in \mathbb{P}$ be a prime number and $k \in \mathbb{Z}$ is an integer, then:*

$$k^p \equiv k \pmod p, \tag{4.4}$$

Proof. □

Remark 4. *Fermats theorem is also often written as the equivalent equation $k^{p-1} \equiv 1 \pmod p$, which can be derived from the original equation by dividing both sides of the congruency by k. In particular this gives a way to compute the multiplicative inverse of a number in modular arithmetics.*

```
sage: ZZ(64)** ZZ(137) % ZZ(137) == ZZ(64)
True
sage: ZZ(64)** ZZ(137-1) % ZZ(137) == ZZ(1)
True
```

Another theorem that is important for doing calculations with congruences is the following Chinese remainder theorem, as it provides a way to solves systems congruency equations (**?** chapter 3.15).

Theorem 4.1.2.3 (Chinese remainder theorem)**.** *For any $k \in \mathbb{N}$ let coprime natural numbers $n_1, \dots n_k \in \mathbb{N}$ as well as the integers $a_1, \dots a_k \in \mathbb{Z}$ be given. Then the so-called simultaneous congruency*

$$\begin{array}{ll} x \equiv a_1 & (\bmod n_1) \\ x \equiv a_2 & (\bmod n_2) \\ & \cdots \\ x \equiv a_k & (\bmod n_k) \end{array} \tag{4.5}$$

has a solution and all possible solutions of this congruence system are congruent modulo $n_1 \cdot \ldots \cdot n_k$.

Proof. (**?** chapter 3.15) □

Remark 5. *From the proof as given in* **?** *chapter 3.15, the following algorithm to find all solutions to any given system of congruences can be derived TODO:WRITE IN ALGORITHM STYLE*

- *Compute $N = n_1 \cdot n_2 \cdot \ldots \cdot n_k$*
- *For each $1 \le j \le k$, compute $N_j = \frac{N}{n_j}$*
- *For each $1 \le j \le k$, use the extended Euklidean algorithm (4.1.1.2) to compute numbers s_j as well as t_j, such that $1 = s_j \cdot n_j + t_j \cdot N_j$ holds.*
- *A solution to the congruency system is then given by $x = \sum_{j=1}^{k} a_j \cdot t_j \cdot N_j$.*
- *Compute $m = x \bmod N$. The set of all possible solutions is then given by $x + m \cdot \mathbb{Z} = \{\dots, x - 2m, x - m, x, x + m, x + 2m, \dots\}$.*

Remark 6. *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 $n_1, \dots, n_k$.*

Example 12. *To illustrate how to solve simultaneous congruences using the Chinese remainder theorem, let's look at the following system of congruencies:*

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

So here 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

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

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 $\{\dots, -2222, -1067, 88, 1243, 2398, \dots\}$. In particular, there are infinitely many different solutions.

```
sage: CRT_list([4,1,3,0], [7,3,5,11])                    62
```

```
88
```

Congruency modulo n is an equivalence relation on the set of integers, where each class is the set of all integers that have the same remainder when divided by n. If then follows from the properties of Euclidean division,that there are exactly n different equivalence classes.

If we go a step further and identify each equivalence class with the corresponding remainder of the Euclidean division, we get a new set, where integer addition and multiplication can be projected to a new kind of addition and multiplication on the equivalence classes.

Roughly speaking the new rules are computed by taking any element of the firsr equivalence class and and element of the second, then add or multiply them in the usual way and see in which equivalence class the result is contained.

The following theorem makes the idea precises

Theorem 4.1.2.4. *Let $n \in \mathbb{N}_{\geq 2}$ be a fixed, natural number and $\mathbb{Z}_n$ the set of equivalence classes of integers with respect to the congruence modulo n relation. Then $\mathbb{Z}_n$ forms a commutative ring with unit with respect to the addition and multiplication defined above.*

Proof. (**?** sentence 1) □

Remark 7. *DESCRIBE NEUTRAL ELEMENTS AND HOW TO ADD, EXPLAIN HOW TO FIND THE NEGATIVE OF A NUMBER AND HOW TO SUBTRACT AND HOW TO MULTIPLY...*

The following example makes the abstract idea more concrete

Example 13 (Arithmetics modulo 6). *Choosing the modulus $n = 6$ we have six equivalence classes of integers which are congruent modulo 6 (which have the same remainder when divided by 6). We write*

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

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 in which equivalence class the result will be.

So we have $14 + (-1) = 13$ and 13 is in the equivalence class (of) 1. Hence in Z_6 we have that $2 + 5 = 1$!

Applying the same reasoning to all equivalence classes, addition and multiplication can be transferred to the equivalence classes and the results are summarized in the following addition and multiplication tables for $\mathbb{Z}_6$:

+	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	2	3	2	1

These two tables are all you need to be able to calculate in $\mathbb{Z}_6$. For example, to determine the multiplicative inverse of a remainder class, look for the entry that results in 1 in the product table. For example the multiplicative inverse of 5 is 5 itself, since $5 \cdot 5 = 1$. Similar to the integers not all numbers have inverses. For example there is no element, that when multiplied with 4 will give 1. However in contrast to what we know from integers, there are non zero numbers, that, when multiplied gives zero (e.g $4 \cdot 4 = 0$).

```
sage: Z6=Integers(6) # Define integers modulo 6
sage: Z6(2)+Z6(5) # standard representatives of a class
1
sage: Z6(14)+Z6(-1) # different representatives for same
   class
1
sage: - Z6(2) # additive inverse
4
sage: Z6(5)**(-1) # multiplicative inverse if exists
5
```

4.1.3 Polynome

Following **?** we want to give a brief introduction to polynomials.

Definition 4.1.3.1 (Polynomials)**.** *Let R be a commutative ring with unit 1. Then we call the expression*

$$\sum_{n=0}^{m} a_n t^n = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m \ , \tag{4.6}$$

*with unknown x and coefficients $a_n \in R$ a **polynomial with coefficients in** R and we write $R[x]$ for the set of all polynomials with coefficients in R.*

We often simply write $P(x) \in R[x]$ for a polynomial and denote the constant term as $P(0)$.

Example 14. *The so-called zero polynomial is the polynomial $0(x) := 0 \cdot x^0$. In analogy to the additively neutral element $0 \in R$ of general rings, we also often write 0 for this polynomial.*

The so-called unit polynomial is the polynomial $1(x) = 1 \cdot x^0$. In analogy to the multiplicatively neutral element $1 \in R$ of a general ring, we also often write 1 for this polynomial.

```
sage: Z6x = Z6['x']
sage: Z6x
Univariate Polynomial Ring in x over Ring of integers modulo
   6
sage: p = Z6x([1,2,3,4])
sage: p
4*x^3 + 3*x^2 + 2*x + 1
```

Definition 4.1.3.2 (degree)**.** *The degree degree$(P(x))$ of a polynomial $P(x) \in R[x]$ is defined as follows: If $P(x)$ is the zero polynomial, we define deg$(P(x)) := -1$. For every other polynomial we define degree$(P(x)) = n$ if a_n is the highest non-vanishing coefficient of $P(x)$.*

```
sage: p.degree()
3
sage: Z6x([0]).degree()
-1
```

Polynomials can be added and multiplied in such a way that the set of all polynomials becomes a commutative ring:

Definition 4.1.3.3. *Let $\sum_{n=0}^{\infty} a_n x^n$ and $\sum_{n=0}^{\infty} b_n x$ be two polynomials from $R[x]$. Then the sum and the product of these polynomials is defined as follows:*

$$\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n \tag{4.7}$$

$$\left(\sum_{n=0}^{\infty} a_n x^n\right) \cdot \left(\sum_{n=0}^{\infty} b_n x^n\right) = \sum_{n=0}^{\infty} \sum_{i=0}^{n} a_i b_{ni} x^n \tag{4.8}$$

```
sage: q = Z6x([5,-3,2,])
sage: p + q
4*x^3 + 5*x^2 + 5*x
sage: p*q
2*x^5 + 3*x^3 + 5*x^2 + x + 5
sage: p^2
4*x^6 + x^4 + 2*x^3 + 4*x^2 + 4*x + 1
```

In the case of polynomials, it is only necessary to note that the degree of the sum is exactly the maximum of the degrees of the summands and that the degree of the product is exactly the sum of the degrees of the factors.

The ring of polynomials shares a lot of properties with the integers. In particular there is also the concept of Euclidean division and the algorithm of long division defined for polynomials.

Definition 4.1.3.4 (Irreducible Polynomial)**.** *TECHNOBOB*

4.1.4 Lagrange interpolation polynomials

4.2 Galois fields

As we have seen in the previous section, modular arithmetics behaves in many ways similar to ordinary arithmetics of integers. But in contrast to arithmetics on integers or rational numbers, we deal with a finite set of elements, which when implemented on computers will not lead to precision problems.

However as we have seen in the last example modular arithmetics is at the same time very different from integer arithmetics as the product of non zero elements can be zero.

In addition it is also different from the arithmetics of rational numbers, as there is often no multiplicative inverse, hence no division defined.

In this section we will see that modular arithmetics behaves very nicely, whenever the modulus is a prime number. In that case the rules of modular arithmetics exactly parallels exactly the well know rules of rational arithmetics, despite the fact that the actually computed numbers are very different.

The resulting structures are the so called prime fields and they are the base for many of the contemporary algebra based cryptographic systems.

Since Galois fields are strongly connectedto prime numbers we start with a short overview of prime numbers and provide few basic properties like the fundamental theorem of arithmetic, which says that every natural number can be represented as a finite product of prime numbers.

The key insight here, is that when the modulus is a prime number, modular arithmetic has a well defined division, that is absent for general moduli.

A prime number $p \in \mathbb{N}$ is a natural number $p \geq 2$, which is divisible by itself and by 1 only. Such a prime number is called *odd* if it is not the number 2. We write $\mathbb{P}$ for the set of all prime numbers and $\mathbb{P}_{\geq 3}$ for the set of all odd prime numbers.

As the Greek mathematician Euclid was able to prove by contradiction in the famous *theorem of Euclid*, no largest prime number exists. The set of all prime numbers is thus infinite **?**.

Since prime numbers are especially natural numbers, they can be ordered according to size, so that one can get the sequence

$$p_n := 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, \dots \tag{4.9}$$

of all prime numbers, which is sequence $A000040$ in OEIS or **?** chapter 1.4). In particular, we can talk about small and large prime numbers.

As the following theorem shows, prime numbers are in a certain sense the basic units from which all other natural numbers are composed:

Theorem 4.2.0.1 (The Fundamental Theorem of Arithmetic)**.** *Let $n \in \mathbb{N}_{\geq 2}$ be a natural number. Then there are prime numbers $p_1, p_2, \dots, p_k \in \mathbb{P}$, such that:*

$$n = p_1 \cdot p_2 \cdot \ldots \cdot p_k \,. \tag{4.10}$$

*Except for permutations in the factors, this representation is unique and is called the **prime factorization** of n.*

Proof. (**?** sentence 6.) □

Remark 8. *An important question is how fast we can compute the prime factorization of a natural number? This is the famous factorization problem. As far as we know, there is no method on a classical Turing machine that is able to compute this representation in polynomial time. The fastest algorithms known today run sub-exponentially, with $\mathcal{O}((1+\epsilon)^n)$ and some $\epsilon > 0$. THe interested reader can find more on this exciting topic in* **?** *Chapter 10.*

Remark 9. *It should be pointed out however hat the American mathematician Peter Williston Shor developed an algorithm in 1994 which can calculate the prime factor representation of a natural number in polynomial time on a quantum computer* **?***.*

The consequence of this is, of course, that cryosystems, which are based on the time complexity of the prime factor problem, are unsafe as soon as practically usable quantum computers are available.

Definition 4.2.0.2 (Prime Fields). *(? example 3.4.4 or ? definition 3.1) Let $p \in \mathbb{P}$ be a prime number. Then we write $(\mathbb{F}, +, \cdot)$ for the set of congruency classes and the induced addition and multiplication as described in theorem (??) and call it the **prime field** of characteristic p.*

Remark 10. *We have seen in (4.1.2.4) how do compute addition, subtraction and multiplication in modular arithmetics. AS prime fields are just a special case where the modulus is a prime number, all this stays the same. In addition we have also seen in example (XXX) that division is not always possible in modular arithmetics. However the key insight here is, that division is well defined when the modulus is a prime number. This means that in a prime field we can indeed define devision.*

To be more precise, division is really just multiplication with the so called multiplicative inverse, which is really just another element, such that the product of both elements is equal to 1. This is well known from fractional numbers, where for example the multiplicative element of say 3 is simply 1/3, since $3 \cdot 1/3 = 1$. Division by 3 is then noth but multiplication by the inverse 1/3. For example $7/3 = 7 \cdot 1/3$.

We can apply the same reasoning when it comes to prime fields and define division as multiplication with the multiplicative inverse, which leads to the question of how to find the multiplicative inverse of an equivalence class $x \in \mathbb{F}_p$ in a prime field.

As with all fields, 0 has no inverse, which implies, that division by zero is undefined. So lets assume $x \neq 0$. Then $gcd(x, p) = 1$, since p is a prime number and therefore has no divisors (see 4.2.0.1).

So we can use the extended Euclidean algorithm (REF) to compute numbers $x^{-1}, t \in \mathbb{Z}$ with $s \cdot x + t \cdot p = 1$, which gives x^{-1} as the multiplicative inverse of x in $\mathbb{F}_p$, since $x^{-1}x \equiv 1 \pmod{p}$.

Example 15. *To summarize the basic aspects of computation in prime fields, lets 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 F_5 is a field all rules like bracketing (distributivity), summing ect. are identical to the rules we learned in school when we where dealing with rational, real or complex numbers.

So we start by evaluating the bracket and get $\left(\frac{2}{3} - 2\right) \cdot 2 = \frac{2}{3} \cdot 2 - 2 \cdot 2 = \frac{2 \cdot 2}{3} - 2 \cdot 2$. Now we evaluate $2 \cdot 2 = 4$, since $(\ mod\, 4, 5) = 4$ and $-(2 \cdot 2) = -4 = 5 - 4 = 1$, since the negative of a number is just the modulus minus the original number. We therefore get $\frac{2 \cdot 2}{3} - 2 \cdot 2 = \frac{4}{3} + 1$.

Now to compute the faction, we need the multiplicative inverse of 3, which is the number, that when multiplies with 3 in $\mathbb{F}_5$ gives 1. So we use the extended Euclidean algorithm to compute

$$x^{-1} \cdot 3 + t \cdot 5 = 1$$

Note that in the Euclidean algorithm the compuations of each t_k is irrelevant here:

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* $\mathbb{F}_5$.

This implies that we can rewrite our original expression into $\frac{4}{3}+1 = 4\cdot2+1 = 3+1 = 4$.

The following important property immediately follows from Fermat's little theorem:

Lemma 4.2.0.3. *Let* $p \in \mathbb{P}$ *be a prime number and* $\mathbb{Z}_p$ *be associated prime field of the characteristic* p*. Then the equations*

$$x^p = x \quad or \quad x^{p-1} = 1. \tag{4.11}$$

holds for all elements $x \in \mathbb{Z}_p$ *with* $x \neq 0$.

In order to give the reader an impression of how prime fields can be seen, we give a full computation of a prime field in the following example

Example 16 (The prime field $\mathbb{Z}_5$)**.** *For* $n = 5$ *we have five equivalence classes of integers which are congruent modulo* 5*. We write*

$0 := \{\ldots, -5, 0, 5, 10, \ldots\}$, $1 := \{\ldots, -4, 1, 6, 11, \ldots\}$, $2 := \{\ldots, -3, 2, 7, 12, \ldots\}$
$3 := \{\ldots, -2, 3, 8, 13, \ldots\}$, $4 := \{\ldots, -1, 4, 9, 14, \ldots\}$

Addition and multiplication can now be transferred to the equivalence classes. This results in the following addition and multiplication tables in $\mathbb{Z}_5$*:*

+	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

These two tables are all you need to be able to calculate in $\mathbb{Z}_5$*. For example, to determine the multiplicative inverse of an element, look for the entry that results in* 1 *in the product table. This is the multiplicative inverse. For example the multiplicative inverse of* 2 *is* 3 *and the multiplicative inverse of* 4 *is* 4 *itself, since* $4 \cdot 4 = 1$.

Definition 4.2.0.4 (The finite bodies)**.** *Let* $p \in \mathbb{N}$ *be a prime number. Then* $\mathbb{K}_p$ *denotes the remainder class body örper* $\mathbb{Z}/p\mathbb{Z}$ *and* $\mathbb{K}_{p^n}$*, for each* $n \in \mathbb{N}$*, the finite K (unique except for isomorphism) örper with* p^n *elements.*

4.2.1 Square Roots

In this part we deal with *square numbers* and *square roots* in prime fields. To do this, we first define what square roots actually are. We roughly follow Chapter 6.5 in **?**.

Definition 4.2.1.1 (quadratic residue and square roots)**.** *Let* $p \in \mathbb{P}$ *a prime number and* $\mathbb{F}_p$ *the associate 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 equation*

$$x^2 = y \tag{4.12}$$

On the other hand, if y *is given and the quadratic equation has no* x *solution, we call* y *as quadratic non-residue. For any* $y \in \mathbb{F}_p$ *we write*

$$\sqrt{y}_{|p} := \{x \in \mathbb{F}_p \mid x^2 = y\} \tag{4.13}$$

for the set of all square roots of y *in the prime field* $\mathbb{F}_n$*. (If* y *is a quadratic non-residue, then* $\sqrt{y}_{|p} = \emptyset$ *and if* $y = 0$*, then* $\sqrt{y}_{|p} = \{0\}$*)*

Remark 11. *The notation $\sqrt{y}_{|n}$ for the root of square residues is not found in textbooks, but it is quite practical to clearly distinguish between roots in different prime fields as the symbol $\sqrt{y}$ is ambiguous and it must also be specified in which prime field this root is actually meant.*

Remark 12. *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 17 (square numbers and roots in $\mathbb{F}_5$). *Let us consider our example (16) again. All square numbers can be found on the main diagonal of the multiplication table. As you can see, in $\mathbb{Z}_5$ only get the square root of* 0, 1 *and* 4 *are non empty sets and we get* $\sqrt{0}_{|5} = \{0\}$, $\sqrt{1}_{|5} = \{1,4\}$, $\sqrt{2}_{|5} = \emptyset$, $\sqrt{3}_{|5} = \emptyset$ *and* $\sqrt{4}_{|5} = \{2,3\}$.

In order to describe whether an element of a prime field is a square number or not, we define (**?** chapter 6.5) the so called Legendre Symbol as its of used in the literature

Definition 4.2.1.2 (Legendre symbol)**.** *Let $p \in \mathbb{P}$ be a prime number and $y \in \mathbb{F}_p$ an element of the associated prime field. Then the so-called Legendre symbol of y is defined as follows:*

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \textit{if } y \textit{ has square roots} \\ -1 & \textit{if } y \textit{ has no square roots} \\ 0 & \textit{if } y = 0 \end{cases} \tag{4.14}$$

Example 18. *If we look again at the example (16) we have the following Legendre symbols*

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

The following theorem gives a simple criterion for calculating the legendre symbol.

Theorem 4.2.1.3 (Euler criterion)**.** *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*

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}}\ . \tag{4.15}$$

Proof. (**?** proposition 83) □

So the question remains how to actually compute square roots in prime field. The following algorithms give a solution

Definition 4.2.1.4 (Tonelli-Shanks algorithm)**.** *Let p be an odd prime number $p \in \mathbb{P}_{\geq 3}$ and y a quadratic residue in $\mathbb{Z}_p$. Then the so-called Tonneli* **?** *and Shanks* **?** *algorithm computes the two square roots of y. It is defined as follows:*

1. *Find $Q, S \in \mathbb{Z}$ with $p - 1 = Q \cdot 2^S$ such that Q is odd.*
2. *Find an arbitrary quadratic non-remainder $z \in \mathbb{Z}_p$.*
3.
```
$M := S,\quad c := z^Q,\quad t := y^Q,\quad R := y^{\frac{Q+1}{2}},\quad M, c, t, R \in \mathbb{Z}_p$
***while*** $t \neq 1$ ***do***
```

```
        Find the smallest $i$ with $0 < i < M$ and $t^{2^i} = 1$
        $b := c^{2^{M-i-1}}$
        $M := i$,   $c := b^2$,   $t := tb^2$,   $R := R \cdot b$
    end while
```

The results are then the square roots $r_1 := R$ and $r_2 := p - R$ of y in $\mathbb{F}_p$.

Remark 13. *The algorithm (4.2.1.4) works in prime fields for any odd prime numbers. From a practical point of view, however, it is efficient only if the prime number is congruent to* 1 *modulo* 4*, since in the other case the formula from the proposition* **??***, which can be calculated more quickly, can be used.*

4.2.2 Exponents and Logarithms

4.2.3 Extension Fields

Eliptic curve pairings often need finite fields that go beyond the finite prime fields.

In fact it is well known, that for every natural number $n \in \mathbb{N}$ there is a field $\mathbb{F}_n$ with n elements, if and only if n is the power of a prime number, i.e. there is a $m \in \mathbb{N}$ with $n = p^m$ for some prime $p \in \mathbb{P}$.

Theorem 4.2.3.1 (Galois Field)**.** *Let $m \in \mathbb{N}$ and $p \in \mathbb{P}$ a prime number. Then there is a field $\mathbb{F}_{p^m}$ of characteristic p, that contains p^n elements.*

We call such a field a **Galois field**. The following algorithm describes the construction of Galois fields (and more general field extensions):

Construction of $\mathbb{F}_{p^m}$

1. Choose a polynomial $P \in \mathbb{F}[t]$ of degree m, that is irreducible.
2. (Field set) of $\mathbb{F}_p$ is the set of all polynomials in $\mathbb{F}[t]$ of degree $< m$, that is
$$\mathbb{F}_{p^m} := \{a_{k-1}t^{k-1} + a_{k-2}t^{k-2} + \ldots + a_1 t + a_0 \mid a_i \in \mathbb{F}_p\}$$
3. (Field Addition) is just addition of the polynomials in the usual way.
4. (Field Multiplication) Multiply the polynomials in the usual way, then compute the long division by P. The remainder is the product.

Remark 14. *In the definition of $\mathbb{F}_{p^m}$, every a_j can have p different values and we have m many of them. This implies that $\mathbb{F}_{p^m}$ contains p^m many elements.*

The construction depends on the choice of an irreducible polynomial, but it can be shown, that all resulting fields are isomorphic, that is they can be transformed into each other, so they are really just different views on the same thing. From an implementations point of view however some choices are better, because they allow for faster computations.

Example 19 (The Galois field $\mathbb{F}_{3^2}$)**.** *In (XXX) we have constructed the prime field $\mathbb{F}_3$. In this example we apply algorithm (XXX) to construct the Galois field $\mathbb{F}_{3^2}$. We start by choosing an irreducibe polynomial of degree* 2 *with coefficients in $\mathbb{F}_3$. We try $P(t) = t^2 + 1$.*

Maybe the fastest way to show that P is indeed irreducible is to just insert all elements from $\mathbb{F}_3$ to see if the result is never zero. WE compute

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

Now the set $\mathbb{F}_{3^2}$ contains all poynomials of degrees lower then two with coefficients in $\mathbb{F}_3$. This is precisely

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

Addition is then defined as normal addition of polynomials. For example $(t+2)+(2t+2) = (1+2)t + (2+2) = 1$. Doing this computation for all elements give 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

From this table, we can 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 $(2t+1) + (t+2) = 0$ and the negative of an element is that other element, such that the sum gives the additive neutral element.

Multiplication then needs a bit more computation, as you multiply the polynomials and then divide the result by P and keep the remainder. For example $(t+2) \cdot (2t+2) = 2t^2 + 2t + t + 1 = 2t^2 + 1$. Long division by $P(t)$ then gives $2t^2 + 1 : t^2 + 1 = 2 + \frac{2}{t^2+1}$, so the remainder is 2 and find that the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$ is 2. Doing this computation for all elements give the following multiplication table (DOTHIS!!! THE TABLE NEEDS AN UPDATE)

·	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

From this table, we can 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 $(2t+1) + (t+2) = 0$ and the negative of an element is that other element, such that the sum gives the additive neutral element.

4.3 Epileptic Curves

In this section we introduce epileptic curves as they are used in cryptography, hwhich are certain types of commutative groups basically, well suited for various constructions of various cryptographic primitives.

The eliptic curves we consider are all defined over Galoise fields, so the reader should be familiar with the contend of the previous section.

Definition 4.3.0.1 (Short Weierstraß elliptic Curve)**.** *Let $\mathbb{F}_q$ be a Galois field and $a, b \in \mathbb{F}_q$ be two field elements with $4a^3+27b^2 \neq 0$. Then a short Weierstrass elliptic curve E/F_q over F_q is defined as the set*

$$E/\mathbb{F}_q = \{(x,y) \in \mathbb{F}_q \times \mathbb{F}_q \mid y^2 = x^3 + ax + b\} \bigcup \{\mathcal{O}\}$$

of all pairs $(x,y) \in \mathbb{F}_q \times \mathbb{F}_q$ of field elements, that satisfy the short Weierstrass equation $y^2 = x^3 + ax + b$, together with the "point at infinity $\mathcal{O}$.

If the characteristic of the Galois field is 2 or 3, that is if $\mathbb{F}_q$ is of the form $\mathbb{F}_{2^n}$ or $\mathbb{F}_{3^n}$ for some $n \in \mathbb{N}$, then this is not the most general way to describe an elliptic curve. However for our purposes this is al we need.

An interesting question is: How many elements does a curve over a finite contain? Since the curve consists of pairs of elements from $\mathbb{F}_q$ plus the point at infinity and $\mathbb{F}_q$ contains q elements, the curve can contain at most $q^2 + 1$ many elements. However the following estimation is more precise:

Theorem 4.3.0.2 (Hasse bound)**.** *Let $E/\mathbb{F}_q$ be an elliptic curve over a finite field $\mathbb{F}_q$ and let $|E/\mathbb{F}_q|$ be the number of elements in that curve. Then there is a number t called the* ***trace****, with $|t| \leq 2\sqrt{q}$ and*

$$|E/\mathbb{F}_q| = q + 1 - t$$

So roughly speaking, the number of elements in an elliptic curve is approximately equal to the sizeq of an underlying field.

Example 20. *Lets consider our prime field F_5 from (XXX). If we choose $a = 1$ and $b = 0$ then $4a^3 + 27b^2 = 4 \neq 0$ and the corresponding elliptic curve $E/\mathbb{F}_3$ is given by all pairs (x,y) from $\mathbb{F}_5$ such that $y^2 = x^3 + x$. We can find this set simply by trying all 25 combinations of pairs. We get*

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

So our elliptic curve contains 4 elements and the trace t is therefore 2.

Example 21. *Consider our prime field F_5 from (XXX). If we choose $a = 1$ and $b = 1$ then $4a^3 + 27b^2 = 1 \neq 0$ and the corresponding elliptic curve $E/\mathbb{F}_3$ is given by all pairs (x,y) from $\mathbb{F}_5$ such that $y^2 = x^3 + x + 1$. We can find this set simply by trying all 25 combinations of pairs. We get*

$$E_2/\mathbb{F}_5 = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\}$$

So our elliptic curve contains 9 elements and the trace t is therefore -3.

4.4 The 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 points together with the point at infinity, which serves as the neutral element.

The origin of this law is geometric and known as the chord-and-tangent rule. The rule can be described in the following way:

- (Point addition) Let $P, Q \in E/\mathbb{F}_q - \{\mathcal{O}\}$ with $P \neq Q$ be two distinct points on an elliptic curve $E/\mathbb{F}_q - \{\mathcal{O}\}$, that are both not the point at infinity. Then one can define the sum of P and Q 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 $R = P + Q$ of P and Q as 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 $\mathcal{O}$.
- (Point doubling) Let $P \in E/\mathbb{F}_q$ with $P \neq Q$ be a points on an elliptic curve $E/\mathbb{F}_q - \{\mathcal{O}\}$. Then the double $2P$ of P is defined as follows: Consider the line wich is tangent to the elliptic curve at P. It can be shown, that it intersects the elliptic curve at the second point R'. $2P$ is then the reflection of this point at the x-axis.
- (Point at infinity) We define $P + \mathcal{O} = \mathcal{O}$ for all points of the eliptic curve $P, Q \in E/\mathbb{F}_q$.

It can be shown that the points of an elliptic curve have a group structure with respect to the tangent and chord rule, with $\mathcal{O}$ as the neutral element. The inverse of any element $P \in E/\mathbb{F}_q$ is the reflection of P on the x-axis.

Translating the chord-and-tanent-rule into algebraic expressions gives the following laws for computing the sum of points on an elliptic curve in short Weierstrass form:

- (Commutativity) $P + Q = Q + P$ for all $P, Q \in E/\mathbb{F}_q$.
- (The neutral element) $P + \mathcal{O} = P$ for all $P \in E/\mathbb{F}_q$.
- (Addition rule) For $P, Q \in E/\mathbb{F}_q$ with $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ and $x_1 \neq x_2$, the sum $R = P + Q$ is given by $R = (x_3, y_3)$ with

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

- (Doubling rule) For $P \in E/\mathbb{F}_q$ with $P = (x, y)$ and $y \neq 0$, the double of P (sum of P with itself) is given by $2P = (x', y')$ with

$$x' = \left(\frac{3x^2 + a}{2y}\right)^2 - 2x \quad y' = \left(\frac{3x^2 + a}{2y}\right)^2 (x - x') - y$$

- (The inverse element) For $P \in E/\mathbb{F}_q$ with $P = (x, y)$, the inverse (negative) element is given by $-P := (x, -y)$.

It can be shown, that when $x_1 = x_2$ then $y_1 = -y_2$. This implies that the previous rules are complete, since in doubling the case $y = 0$, means that the point is its own inverse and hence doubling gives the point at infinity.

As we can see, it is very efficient to compute inverses on elliptic curves.

Definition 4.4.0.1 (Elliptic curve exponentiation and generators). *Let $E/\mathbb{F}_q$ an elliptic curve and $P \in E/\mathbb{F}_q$ a point on that curve. Then the* ***elliptic curve exponentiation*** *with base P is given by*

$$[\cdot]P : \mathbb{Z} \to E/\mathbb{F}_q; m \mapsto [m]P$$

where $[m]P = P + P + \ldots + P$ is the m-fold sum of P with itself. Moreover the point P is called a ***generator*** *of the curve, if $[\mathbb{Z}]P = E/\mathbb{F}_q$.*

Remark 15. *The term exponentiation come from the fact, that if the group law is written in multiplicative notation, then an m-fold product $x \cdot x \cdot \ldots \cdot x$ is usually written in as x^m. So our exponential map, is an adoptation of that for groups with an additive notation of the group law.*

Example 22. *Lets consider the curve $E_1/\mathbb{F}_5$ from example XXX again. We have*

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

So as always $\mathcal{O}$ is the neutral element. Since all elements have 0 as their y-coordinate, it follows that all of them are self inverse, that is $-P = P$. To add, say $(2,0)$ and $(3,0)$ we use the addition rule, since their x-coordinates differ, we get

$$(0,0) + (2,0) = (3,0)$$
$$(0,0) + (3,0) = (2,0)$$
$$(2,0) + (3,0) = (0,0)$$

As we can see $(0,2)$ is not a generator of the group since $[1](0,2) = (0,2)$, $[2](0,2) = (0,2) + (0,2) = \mathcal{O}$. HMM I GUESS THERE IS SOMETHING WRONG HERE. THREE TWO ELEMENT SUBGROUPS SHOULDN'T EXIST??

Example 23. *Consider our prime field F_5 from (XXX). If we choose $a = 1$ and $b = 1$ then $4a^3 + 27b^2 = 1 \neq 0$ and the corresponding elliptic curve $E/\mathbb{F}_3$ is given by all pairs (x,y) from $\mathbb{F}_5$ such that $y^2 = x^3 + x + 1$. We can find this set simply by trying all 25 combinations of pairs. We get*

$$E_2/\mathbb{F}_5 = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\}$$

So our elliptic curve contains 9 elements and the trace t is therefore -3.

Definition 4.4.0.2 (The elliptic curve discrete logarithm). . *Let $E/\mathbb{F}_q$ be an elliptic curve and $P, Q \in E/\mathbb{F}_q$ be two points. Then the elliptic curve discrete logarithm problem consists of finding a solution $m \in \mathbb{Z}$, such that*

$$P = [m]Q$$

Such an equation is also called a discrete logarithm relation between P and Q

Remark 16. *If neither P nor Q is a generator of a curve, then a discrete logarithm relation does not exists. On the other hand if one of the elements is a generator, then infinite many solutions exists.*

STUFF ON THE INFEASABILITY TO COMPUTE DISCRETE LOGARITHMS

Definition 4.4.0.3 (Cofactor Clearing). *Since $BLS6-6(13)$ is a subgroup on our curve, it is not possible to leave the subgroup using the curves algebraic laws like scalar multiplication or addition. However in applications it often happens that random elements of the curve are generated, while what we really want are points in the subgroup. To get those points we can use cofactor clearing.*

Cryptographically secure elliptic curves Not all elliptic curves satisfy the requirements from applied cryptography .... Here is a list of properties a curve should satisfy:

1. TECHNOBOB

4.5 Constructing Elliptic curves

4.5.1 The Complex Multiplication Method

- Choose a prime number $p \in \mathbb{P}$ and integers $t, D \in \mathbb{Z}$, such that the equation $-Dv^2 = 4p - t^2$ has solutions $\pm v \in \mathbb{Z}$.
- If one of the values $p + 1 - t$ or $p + 1 + t$ a prime number, then proceed to the next steps, otherwise we go back to step 1.
- Compute the set $CL(Dv^2) = \{(a, b, c) \mid a, b, c \in \mathbb{Z}, |b| \leq a \leq \sqrt{\frac{Dv2}{3}}, a \leq c, b^2 - 4ac = -Dv^2, (a, b, c) = 1WHATISTHIS?\}$. If $|b| = a$ or $a = c$, then $b \geq 0$.
- Compute $H_D(x) = \Pi_{(a,b,c)\in CL(D)}(x - j(\frac{-b+\sqrt{-Dv^2}}{2a}))$
- Round the coefficients of H_D to the closed integers.
- Compute $H_{D,p} = H_D\ mod_p$
- Find a root j of $H_{D,p}$
- If $j \neq 0$ or $j \neq 1728$, then choose $c \in \mathbb{F}_p$ and define the elliptic curve $E/\mathbb{F}_p$ defined by $y^2 = x^3 + 3kx + 2k$ for $k = \frac{j}{1728-j}$.
- Compute the order of $E/\mathbb{F}_p$. If it divides either $p + 1 - u$ or $p + 1 - u$, then $E/\mathbb{F}_p$ is the result.
- Otherwise choose $c \in \mathbb{F}_p$ with $c \neq 1$ and $c \neq 0$ and define the elliptic curve $E'/\mathbb{F}_p$ defined by $y^2 = x^3 + 3kc^2x + 2kc^3$ for $k = \frac{j}{1728-j}$.

4.6 Classes of elliptic curves

4.6.1 BLS6-6 – our pen& paper curve

In this example 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&paper examples, which mostly means that the curve should contain only a few points, such that we are able to derive exhaustive addition and pairing tables.

A well understood family of pairing friendly curves are the BLS curves (STUFF ABOUT THE HISTORY AND THE NAMING CONVENTION)

BLS curves are particular useful in our case if the embedding degree k satisfies $k \equiv 6 \pmod 0$. In this case the system of polynomials from section XXX parameterizes these curves.

Of course the smallest embedding degree k that satisfies the congruency, is $k = 6$. We therefore aim for a BLS6 curve as our main pen&paper example.

As explained in XXX, the defining polynomials for any BLS6 curve are given by

$$\begin{aligned} r(x) &= \Phi_6(x) \\ t(x) &= x + 1 \\ q(x) &= \frac{1}{3}(x-1)^2(x^2 - x + 1) + x \end{aligned}$$

where Φ_6 is the 6-th cyclotomic polynomial. For any $x \in \mathbb{N}$, where $r(x), t(x), q(x)$ are natural numbers, with $q(x) > 3$ and $r(x) > 3$ those values describe elliptic curves with discriminant $D = 3$, characteristic $p(x)$, prime order subgroup $r(x)$ and Frobenious trace $t(x)$.

We start by looking-up the 6-th cyclotomic polynimial which is $\Phi_6 = x^2 - x + 1$ and then insert small values for x into the defining polynomials r, t, q. This gives the following results:

$x = 1$	$(r(x), t(x), q(x))$	$(1, 2, 1)$
$x = 2$	$(r(x), t(x), q(x))$	$(3, 3, 3)$
$x = 3$	$(r(x), t(x), q(x))$	$(7, 4, \frac{37}{3})$
$x = 4$	$(r(x), t(x), q(x))$	$(13, 5, 43)$

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 use $x = 4$, which defines a curve of fields of characteristic 43. Since the prime field $\mathbb{F}_{43}$ has 43 elements and 43 has binary representation 101011, which are 6 digits, the name of our pen&paper curve should be $BLS6 - 6$.

We can check that the embedding degree is indeed 6, since $k = 6$ is the smallest number k such that $r = 13$ divides $43^k - 1$.

Strictly speaking BLS6-6 is not pairing friendly according to the definition in XXX, since indeed $r = 13 > \sqrt{43}$, but the second requirement is not satisfied. This however is irrelevant as the hole point of constructing this curve is to have a "large" prime oder subgroup that is as small as possible.

From the the defining equations of BLS curves, we can immediately deduce that BLS6-6 has a "large" subgroup of prime order 13, which is well suited for our purposes as 13 elements can be easily handled in the associated addition, scalar multiplication and pairing tables.

To see how the rest of the curve will look, we use XX to compute the number of rational points on the curve which is either $q + 1 - t$ or $q + 1 + t$. We get $43 + 1 - 5 = 39$ or $43 + 1 + 5 = 49$. Since our subgroup order $r = 13$ must divide the number of points, it follows that our curve has 39 point and hence a cofactor of 3, which implies that there is a single non trivial "small" order subgroup that contains three elements.

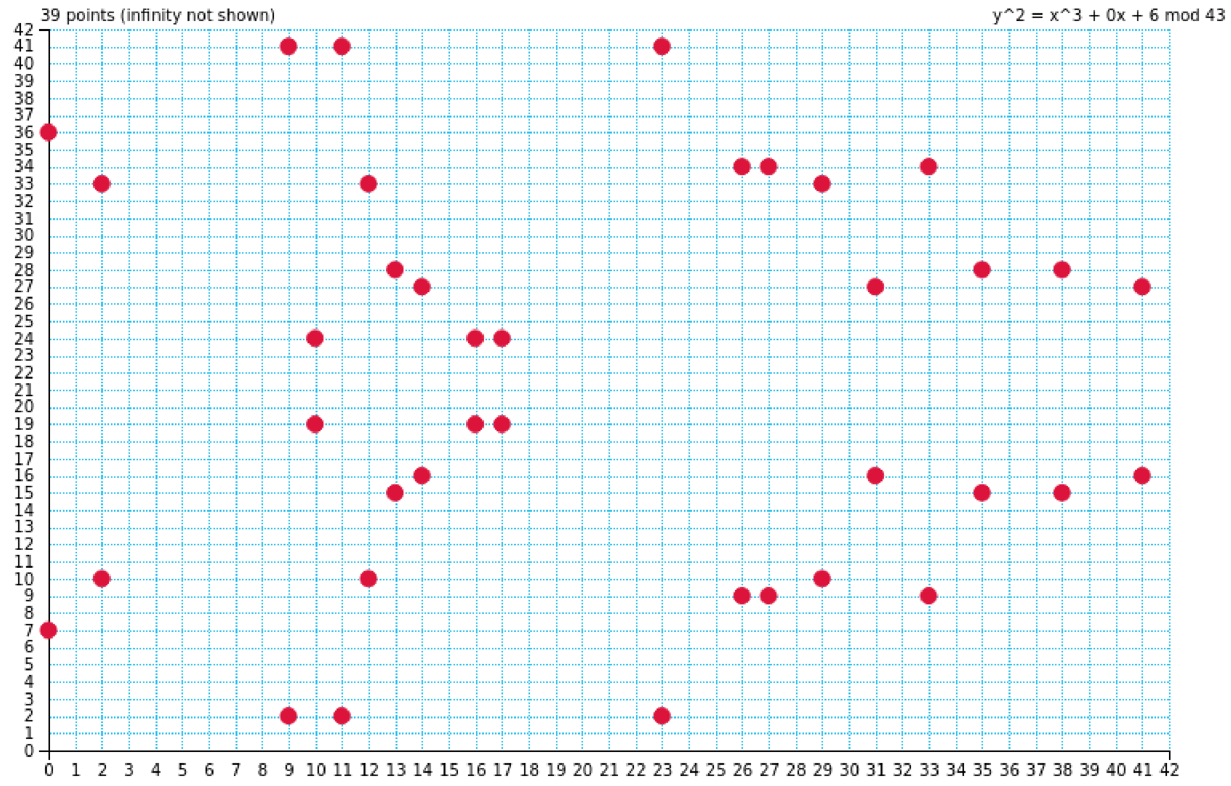
To compute the defining equation $y^2 = x^3 + ax + b$ of BLS6-6, we use the complex multiplication algorithm as described above in XXX. The goal is to find $a, b \in \mathbb{F}_{43}$ representations, that are particulary nice to work with. As shown for example in XXX the discriminant D of all BLS curves is -3, which gives them the general form $y^2 = x^3 + b$.

This is because the Hilbert class polynomial $H_3(x) = x$, since $CL(3) = \{[1, 1, 1]\}$ and in this case $j(\frac{-1+i\sqrt{3}}{1}) = 0$. It follows that the general curve equation is given by $y^2 = x^3 + b$ and it only remains to find b, such that the curve has the correct number of points which is 39. Since $b \in \mathbb{F}_{43}$, we can just put values for b into the equation and count points. The smallest value then is b and we get

$$BLS6-6: \quad y^2 = x^3 + 6 \quad \text{for all } x, y \in \mathbb{F}_{43}$$

There are other choice for b like $b = 10$ or $b = 23$, but all these curves are isomorphic and hence represent the same thing really but in different way only.

Since BLS6-6 only contains 39 points it is possible to give a visual impression of the



curve:

As we can see our curve is somewhat nice, as 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.

Note: Is there a way to printe the entire addition table from https://graui.de/code/elliptic2/ here? Would be nice to have but is a bit large.

Since the order of BLS6-6 is $39 = 3 \cdot 13$, we know that it has a "large" subgroup of order 13 and small subgroup of order 3. We can use XXX to find those groups. We have $BLS6-6(3) = \{\mathcal{O}, (0,7), (0,36)\}$.

In addition we have the generator $g_{BLS6} := (13,15)$ that generates

$$\begin{aligned} BLS6-6(13) = & \\ & \{(13,15) \to (33,34) \to (38,15) \to (35,28) \to (26,34) \to (27,34) \to \\ & (27,9) \to (26,9) \to (35,15) \to (38,28) \to (33,9) \to (13,28) \to \mathcal{O}\} \end{aligned} \tag{4.16}$$

Computations "in the exponent": In cryptography and in particular in snarks a lot HAPPENS IN THE EXPONENT...

To use our example to explain what this means observe that from this representation, we can deduce a map from the scalar field $\mathbb{F}_{13}$ to $BLS6-6(13)$ with respect to our generator. WE have

$$[\cdot]_{(13,15)} : \mathbb{F}_{13} \to BLS6-6(13) \; ; \; x \mapsto [x](13,15)$$

So for example we have $[1]_{(13,15)} = (13,15)$, $[7]_{(13,15)} = (27,9)$ and $[0]_{(13,15)} = \mathcal{O}$. In particular this map is a homomorphism of groups from the additive group $\mathbb{F}_{13}$ to $BLS6-6(13)$. This means in particular, the the additive neutral element from $\mathbb{F}_{13}$ is mapped to $\mathcal{O}$ and negatives are mapped to inverses. For example $[-2]_{(13,15)} = -[2]_{(13,15)}$, since $[-2]_{(13,15)} = [11]_{(13,15)} = (33,9) = (33,-34) = -(33,34) = -[2]_{(13,15)}$

The map also give a visualization of the ECDL problem in $BLS6-6(13)$, which is concerned with finding solutions $x \in \mathbb{F}_{13}$ for the equation $[x]_{(13,15)} = (x,y)$ for any $(x,y) \in BLS6-6(13)$. Of course ECDL is not hard in $BLS6-6(13)$, since we can deduce the solutions easily from XXX. For example the solution to $[x]_{(13,15)} = (35,15)$ is $x = 9$, since $[9](13,15) = (35,15)$.

Since $[0]_{(13,15)}$ maps the group of cyclic integers modulo 13 onto our group $BLS6-6(13)$, we can use this to write down the group law in the following way:

$\cdot$	$\mathcal{O}$	(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)

Cofactor clearing:

Given an arbitrary point on the curve that is not in any of our two subgroups like $(2,33)$, we can project it on both subgroups $BLS6-6(3)$ and $BLS6-6(13)$ respectively, by *multiplication with the cofactor*. Since $39 = 3 \cdot 13$, we have to multiply $(2,33)$ with 13 to map it onto $BLS6-6(3)$ and we have to multiply $(2,33)$ with 3 to map it onto $BLS6-6(13)$. Indeed we get $[13](2,33) = (0,36)$ which is an element of $BLS6-6(3)$ and $[3](2,33) = (35,15)$ which is an element of $BLS6-6(13)$

MNT-Curve Cycles

Chapter 5

Zk-Proof Systems

Some philosophical stuff about compuational models for snarks. Bounded computability...

5.1 Computational Models

Proofs are the evidence of correctness of the assertions, and people can verify the correctness by reading the proof. However, we obtain much more than the correctness itself:After you read one proof of an assertion, you know not only the correctness, but also why itis correct. Is it possible to solely show the correctness of an assertion without revealing theknowledge of proofs? It turns out that it is indeed possible, and this is the topic of today'slecture: Zero Knowledge Systems.

Example 24 (Generalized factorization snark)**.** *As one of our major running examples we want to derive a zk-SNARK for the following generalized factorization problem:*

Given two numbers $a, b \in \mathbb{F}_{13}$, find two additional numbers $x, y \in \mathbb{F}_{13}$, such that

$$(x \cdot y) \cdot a = b$$

and proof knowledge of those numbers, without actually revealing them.

Of course this example reduces to the classic factorization problem (over $\mathbb{F}_{13}$ by setting $y = 1$)

This zero knowledge system deals with the following situation: "Given two publicly known numbers $a, b \in \mathbb{F}_{13}$ a proofer can show that they know two additional numbers $x, y \in \mathbb{F}_{13}$, such that $(x \cdot y) \cdot a = b$, without actually revealing x or y."

Of course our choice of what information to hide and what to reveal was completely arbitrary. Every other split would also be possible, but eventually gives a different problem.

For example the task could be to not hide any of the variables. Such a system has no zero knowledge and deals with verifiable computations: "A worker can proof that they multiplied three publicly known numbers $a, b, x \in \mathbb{F}_{13}$ and that the result is $z \in \mathbb{F}_{13}$, in such a way that no verifier has to repeat the computation."

5.1.1 Formal Languages

Roughly speaking a formal language is nothing but a set of words, that are strings of letters taken from some alphabet and formed according to some defining rules of that language.

In computer science, formal languages are used for defining the grammar of programming languages in which the words of the language represent concepts that are associated with particular meanings or semantics. In computational complexity theory, decision problems are typically defined as formal languages, and complexity classes are defined as the sets of the formal languages that can be parsed by machines with limited computational power.

Definition 5.1.1.1 (Formal Language). *Let Σ be a set and Σ^* the set of all finite strings of elements from Σ. Then a **formal language** L is a subset of Σ^*. The set Σ is called the **alphabet** of L and elements from L are called **words**. The rules that specify which strings from Σ^* belong to L are called the **grammar** of L.*

*In the context of proofing systems we often call words **statements**.*

Example 25 (Generalized factorization snark). *Consider example 24 again. Definition 5.1.1.1 is not quite suitable yet to define the example, since there is not distinction between public input and private input.*

However if we assume for the moment that the task in example 24 is to simply find $a, b, x, y \in \mathbb{F}_{13}$ such that that $x \cdot y \cdot a \cdot = b$, then we can define the entire solution set as a language L_{factor} over the alphabet $\Sigma = \mathbb{F}_{13}$. We then say that a string $w \in \Sigma^*$ is a statement in our language L_{factor} if and only if w consists of 4 letters w_1, w_2, w_3, w_4 that satisfy the equation $w_1 \cdot w_2 \cdot w_3 = w_4$.

Example 26 (Binary strings). *If we take the set $\{0, 1\}$ as our alphabet Σ and imply no rules at all to form words in this set. Then our language L is the set $\{0, 1\}^*$ of all finite binary strings. So for example $(0, 0, 1, 0, 1, 0, 1, 1, 0)$ is a word in this language.*

Example 27 (Programing Language).

Example 28 (Compiler).

As we have seen in general not all strings from an alphabet are words in a language. So an important question is, weather a given string belongs to a language or not.

Definition 5.1.1.2 (Relation, Statement, Instance and Witness). *Let Σ_I and Σ_W be two alphabets. Then the binary relation $R \subset \Sigma_I^* \times \Sigma_W^*$ is called a **checking relation** for the language*

$$L_R := \{(i, w) \in \Sigma_I^* \times \Sigma_W^* \;|R(i, w)\;\}$$

*of all **instances** $i \in \Sigma_I^*$ and **witnesses** $i \in \Sigma_I^*$, such that the **statement** (i, w) satisfies the checking relation.*

Remark 17. *To summarize the definition, a statement is nothing but a membership claim of the form $x \in L$. So statements are really nothing but strings in an alphabet that adhere to the rules of a language.*

However in the context of checking relations, there is another interpretations in terms of a knowledge claim of the form "In the scope of relation R, I know a witness for instance x." This is of particular importance in the context of zero knowledge proofing systems, where the instance represents public knowledge, while the witness represents the data that is hidden (the zero-knowledge part).

For some cases, the knowledge and membership types of statements can be informally considered interchangeable, but formally there are technical reasons to distinguish between the two notions (See for example XXX)

Example 29 (Generalized factorization snark). *Consider example 24 and our associate formal language 25. We can define another language $L_{zk-factor}$ for that example by defining the alphabet $\Sigma_I \times \Sigma_W$ to be $\mathbb{F}_{13} \times \mathbb{F}_{13}$ and the checking relation $R_{zk-factor}$ such that $R(i,w)$ holds if and only if instance i is a two letter string $i = (a,b)$ and witness w is a two letter string $w = (x,y)$, such that the equation $x \cdot y \cdot a = b$ holds.*

So to summarize four elements $x, y, a, b \in \mathbb{F}_{13}$ form a statement $((x,y),(a,b))$ in $L_{zk-factor}$ with instance (a,b) and witness x, y, precisely if, given a and b, the values x and y are a solution to the generalized factorization problem $x \cdot y \cdot a = b$.

Example 30 (SHA256 relation). *ssss*

As the following example shows checking relations and their languages are quite general and able to express in particular the class of all terminating computer programs:

Example 31 (Computer Program). *Let A be a terminating algorithm that transforms a binary string of inputs in finite execution steps into a binary output string. We can then interpret A as a map*

$$A : \{0,1\}^* \to \{0,1\}^*$$

Algorithm A then defines a relation $R \subset \{0,1\}^* \times \{0,1\}^*$ in the following way: instance string $i \in \{0,1\}^*$ and witness string $w \in \{0,1\}^*$ satisfy the relation R, that is $R(i,w)$, if and only if w is the result of algorithm A executed on input instance i.

5.1.2 Circuits

Definition 5.1.2.1 (Circuits). *Let Σ_I and Σ_W be two alphabets. Then a directed, acyclic graph C is called a* ***circuit*** *over $\Sigma_I \times \Sigma_W$, if the graph has an ordering and every node has a label in the following way:*

- *Every source node (called input) has a letter from $\Sigma_I \times \Sigma_W$ as label.*
- *Every sink node (called output) has a letter from $\Sigma_I \times \Sigma_W$ as label.*
- *Every other node (called gate) with j incoming edges has a label that consist of a function $f : (\Sigma_I \times \Sigma_W)^j \to \Sigma_I \times \Sigma_W$.*

Remark 18 (Circuit-SAT). *Every circuit with n input nodes and m output nodes can be seen a function that transforms strings of size n from $\Sigma_I \times \Sigma_W$ into strings of size m over the same alphabet. The transformation is done by sending the strings from a node along the outgoing edges to other nodes. If those nodes are gates, then the string is transformed according to the label.*

By executing the previous transformation, every node of a circuit has an associated letter from $\Sigma_I \times \Sigma_W$ and this defines a checking relation over $\Sigma_I^* \times \Sigma_W^*$. To be more precise, let C be a circuit with n nodes and $(i,w) \in \Sigma_I^j \times \Sigma_W^k$ a string. Then $R_C(i,w)$ iff THE CIRCUIT IS SATISFIED WHEN ALL LABELS ARE ASSOCIATED TO ALL NODES IN THE CIRCUIT.... BUT MORE PRECISE

MODULO ERRORS. TO BE CONTINUED.....

An Assignment associates field elements to all edges (indices) in an algebraic circuit. An Assignment is valid, if the field element arise from executing the circuit. Every other assignment is invalid.

The checking relation for circuit-SAT then is satidfied if valid asignment (TODO: THE WITNESS/INSTANCE SPLITTING)

Valid assignments are proofs for proper circuit execution.

So to summarize, algebraic circuits (over a field $\mathbb{F}$) are directed acyclic graphs, that express arbitrary, but bounded computation. Vertices with only outgoing edges (leafs, sources) represent inputs to the computation, vertices with only ingoing edges (roots, sinks) represent outputs from the computation and internal vertices represent field operations (Either addition or multiplication). It should be noted however that there are many circuits that can represent the same laguage...

Circuits have a notion of execution, where input values are send from leafs along edges, through internal vertices to roots.

Remark 19. *Algebraic circuits are usually derived by Compilers, that transform higher languages to circuits. An example of such a compiler is XXX. Note: Different Compiler give very different circuit representations and Compiler optimization is important.*

Example 32 (Generalized factorization snark)**.** *Consider our generalized factorization example 24 with associated language 29.*

To write this example in circuit-SAT, consider the following function

$$f : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto (x_1 \cdot x_2) \cdot x_3$$

A valid circuit for $f : \mathbb{F}_{11} \times \mathbb{F}_{11} \times \mathbb{F}_{11} \to \mathbb{F}_{11}; (x_1, x_2, x_3) \mapsto (x_1 \cdot x_2) \cdot x_3$ *is given by:*

$$\begin{tikzcd}
\star \arrow[dr, "in_1"] & & \star \arrow[dl, "in_2"'] & & \\
& \star_{m_1} \arrow[drr, "mid_1"] & & & \star \arrow[dl, "in_3"'] \\
& & & \star_{m_2} \arrow[d, "out_1"'] & \\
& & & \star &
\end{tikzcd}$$

with edge-index set $I := \{in_1, in_2, in_3, mid_1, out_1\}$.

To given a valid assignment, consider the set $I_{valid} := \{in_1, in_2, in_3, mid_1, out_1\} = \{2, 3, 4, 6, 10\}$

$$\begin{tikzcd}
\star \arrow[dr, "2"] & & \star \arrow[dl, "3"'] & & \\
& \star_{m_1} \arrow[drr, "6"] & & & \star \arrow[dl, "4"'] \\
& & & \star_{m_2} \arrow[d, "10"'] & \\
& & & \star &
\end{tikzcd}$$

Appears from multiplying the input values at m_1, m_2 *in* $\mathbb{F}_{13}$, *hence by executing the circuit.*

Non valid assignment: $I_{err} := \{in_1, in_2, in_3, mid_1, out_1\} = \{2, 3, 4, 7, 8\}$

$$\begin{tikzcd}
\star \arrow[dr, "2"] & & \star \arrow[dl, "3"'] & & \\
& \star_{m_1} \arrow[drr, "7"] & & & \star \arrow[dl, "4"'] \\
& & & \star_{m_2} \arrow[d, "8"'] & \\
& & & \star &
\end{tikzcd}$$

Can not appear from multiplying the input values at m_1, m_2 in $\mathbb{F}_{13}$

To match the requirements of the inital task 24, we have to split the statement into instance and witness. So given index set $I := \{in_1, in_2, in_3, mid_1, out_1\}$, we assume that every step in the computation other then in_3 and out_1 are part of the witness. So we choose:

- *Instance $S = \{in_3, out_1\}$.*
- *Witness $W = \{in_1, in_2, mid_1\}$.*

Example 33 (Boolean Circuits).

Example 34 (Algebraic (Aithmetic) Circuits).

Any program can be reduced to an arithmetic circuit (a circuit that contains only addition and multiplication gates). A particular reduction can be found for example in [BSCG+13]

5.1.3 Rank-1 Constraint Systems

Definition 5.1.3.1 (Rank-1 Constraint system). *Let $\mathbb{F}$ be a Galois field, i, j, k three numbers and A, B and C three $(i + j + 1) \times k$ matrices with coefficients in $\mathbb{F}$. Then any vector $x = (1, \phi, w) \in \mathbb{F}^{1+i+j}$ that satisfies the* ***rank-1 constraint system*** *(R1CS)*

$$Ax \odot Bx = Cx$$

(where $\odot$ is the Hadamard/Schur product) is called a ***statement*** *of that system, with* ***instance*** *ϕ and* ***witness*** *w.*

We call k the ***number of constraints****, i the* ***instance*** *size and j the* ***witness*** *size.*

Remark 20. *Any Rank-1 constraint system defines a formal language in the following way: Consider the alphabets $\Sigma_I := \mathbb{F}$ and $\Sigma_W : \mathbb{F}$. Then a checking relation $R_{R1CS} \subset \Sigma_I^i \times \Sigma_W^j \subset \Sigma_I^* \times \Sigma_W^*$ is defined by*

$$R_{R1CS}(i, w) \Leftrightarrow (i, w) \text{ satisfies the R1CS}$$

As shown in XXX such a checking relation defines a formal language. We call this language ***R1CS satisfiability****.*

Example 35 (Generalized factorization snark). *Defining the 5-dimensional affine vector $w = (1, in_1, in_2, in_3, m_1, out_1)$ for $in_1, in_2, in_3, m_1, out_1 \in \mathbb{F}_{13}$ and the $6\times?$-matrices*

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We can instantiate the general R1CS equation $Aw \odot Bw = Cw$ as

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix} \odot \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix}$$

So evaluating all three matrix products and the Hadarmat prodoct we get two constraint equations

$$\begin{aligned} in_1 \cdot in_2 &= m_1 \\ m_1 \cdot in_3 &= out_1 \end{aligned}$$

5.1.4 Quadratic Arithmetic Programs

As shown by [Pinocchio] rank-1 constraint systems can be transformed into so called quadratic arithmetic programs assuming $\mathbb{F}$.

taken from the pinocchio paper. For proving arithmetic circuit-sat. Given a R1CS QAPs transform potential solution vectors into two polynomials p and t, such that p is divisible by t if and only if the vector is a solution to the R1CS.

They are major building blocks for **succinct** proofs, since with high probability, the divisibility check can be performed in a single point of those polynomials. So computationally expensive polynomial division check is reduced TO WHAT? (IN FIELDS THERE IS ALWAYS DIVISIBILITY)

Definition 5.1.4.1 (Quadratic Arithmetic Program). *Assume we have a Galois field $\mathbb{F}$, three numbers i, j, k as well as three $(i + j + 1) \times k$ matrices A, B and C with coefficients in $\mathbb{F}$ that define the R1CS $Ax \odot Bx = Cx$ for some statement $x = (1, i, w)$ and let $m_1, \ldots, m_k \in \mathbb{F}$ be arbitrary field elements.*

*Then a **quadratic arithmetic program** of the R1CS is the following set of polynomials over $\mathbb{F}$*

$$QAP = \left\{ t \in \mathbb{F}[x], \{a_h, b_h, c_h \in \mathbb{F}[x]\}_{h=1}^{i+j+1} \right\}$$

*where $t(x) := \Pi_{l=1}^{k}(x - m_l)$ is a polynomial f degree k, called the **target polynomial** of the QAP and $a_h(x)$, $b_h(x)$ as well as $c_h(x)$ are the unique degree $k - 1$ polynomials that are defined by the equations*

$$a_h(m_l) = A_{h,l} \quad b_h(m_l) = B_{h,l} \quad c_h(m_l) = C_{h,l} \quad h = 1, \ldots, i + j + 1, l = 1, \ldots, k$$

The major point is that R1CS-sat can be reformulated into the divisibility of a polynomials defined by any QAP.

Theorem 5.1.4.2. *Assume that an R1CS and an associated QAP as defined in XXX are given. Then the affine vector $y = (1, i, w)$ is a solution to the R1CS, if and only if the polynomial*

$$p(x) = \left(\sum y_h \cdot a_h(x)\right) \cdot \left(\sum y_h \cdot b_h(x)\right) - \sum y_h \cdot c_h(x)$$

is divisible by the target polynomial t.

The polynomials a_h, b_h and c_h are uniquely defined by the equations in XXX. However to actually compute them we need some algorithm like the Langrange XXX from XXX.

Example 36 (Generalized factorization snark)**.** *In this example we want to transform the R1CS from example 29 into an associated QAP.*

We start by choosing an arbitrary field element for every constraint in the R1CS, since we have 2 *constraints we choose* $m_1 = 5$ *and* $m_2 = 7$

With this choice we get the target polynomial $t(x) = (x-m_1)(x-m_2) = (x-5)(x-7) = (x+8)(x+6) = x^2 + x + 9$.

Since our statement has structure $w = (1, in_1, in_2, in_3, m_1, out_1)$ *we have to compute the following degree* 1 *polynomials*

$\{a_c, a_{in_1}, a_{in_2}, a_{in_3}, a_{mid_1}, a_{out}\}$ $\{b_c, b_{in_1}, b_{in_2}, b_{in_3}, b_{mid_1}, b_{out}\}$ $\{c_c, c_{in_1}, c_{in_2}, c_{in_3}, c_{mid_1}, c_{out}\}$

Apply QAP rule XXX to the $a_{k\in I}$ *polynomials gives*

$$\begin{array}{llllll}
a_c(5) = 0, & a_{in_1}(5) = 1, & a_{in_2}(5) = 0, & a_{in_3}(5) = 0, & a_{mid_1}(5) = 0, & a_{out}(5) = 0 \\
a_c(7) = 0, & a_{in_1}(7) = 0, & a_{in_2}(7) = 0, & a_{in_3}(7) = 0, & a_{mid_1}(7) = 1, & a_{out}(7) = 0
\end{array}$$

$$\begin{array}{llllll}
b_c(5) = 0, & b_{in_1}(5) = 0, & b_{in_2}(5) = 1, & b_{in_3}(5) = 0, & b_{mid_1}(5) = 0, & b_{out}(5) = 0 \\
b_c(7) = 0, & b_{in_1}(7) = 0, & b_{in_2}(7) = 0, & b_{in_3}(7) = 1, & b_{mid_1}(7) = 0, & b_{out}(7) = 0
\end{array}$$

$$\begin{array}{llllll}
c_c(5) = 0, & c_{in_1}(5) = 0, & c_{in_2}(5) = 0, & c_{in_3}(5) = 0, & c_{mid_1}(5) = 1, & c_{out}(5) = 0 \\
c_c(7) = 0, & c_{in_1}(7) = 0, & c_{in_2}(7) = 0, & c_{in_3}(7) = 0, & c_{mid_1}(7) = 0, & c_{out}(7) = 1
\end{array}$$

Since our polynomials are of degree 1 *only we don't have to invoke Langrange method but can deduce the solutions right away.*

Polynomials are defined on the two values 5 *and* 7 *here. Linear Polynomial* $f(x) = m \cdot x + b$ *is fully determined by this. Derive the general equation:*

- $5m + b = f(5)$ *and* $7m + b = f(7)$
- $b = f(5) - 5m$ *and* $b = f(7) - 7m$
- $b = f(5) + 8m$ *and* $b = f(7) + 6m$
- $f(5) + 8m = f(7) + 6m$
- $8m - 6m = f(7) - f(5)$
- $2m = f(7) + 12f(5)$
- $7 \cdot 2m = 7(f(7) + 12f(5))$
- $m = 7(f(7) + 12f(5))$
-
- $b = f(5) + 8m$
- $b = f(5) + 8 \cdot (7(f(7) + 12f(5)))$
- $b = f(5) + 4(f(7) + 12f(5))$
- $b = f(5) + 4f(7) + 9f(5)$

- $b = 10f(5) + 4f(7)$

Gives the general equation: $f(x) = 7(f(7) + 12f(5))x + 10f(5) + 4f(7)$

For a_{in_1} *the computation looks like this:*

- $a_{in_1}(x) = 7(a_{in_1}(7) + 12a_{in_1}(5))x + 10a_{in_1}(5) + 4a_{in_1}(7) =$
- $7(0 + 12 \cdot 1)x + 10 \cdot 1 + 4 \cdot 0 =$
- $7 \cdot 12x + 10 =$
- $6x + 10$
- $a_{mid_1}(x) = 7(a_{mid_1}(7) + 12a_{mid_1}(5))x + 10a_{mid_1}(5) + 4a_{mid_1}(7) =$
- $7(1 + 12 \cdot 0)x + 10 \cdot 0 + 4 \cdot 1 =$
- $7 \cdot 1x + 4 =$
- $7x + 4$

$a_c(x) = 0$	$b_c(x) = 0$	$c_c(x) = 0$
$a_{in_1}(x) = 6x + 10$	$b_{in_1}(x) = 0$	$c_{in_1}(x) = 0$
$a_{in_2}(x) = 0$	$b_{in_2}(x) = 6x + 10$	$c_{in_2}(x) = 0$
$a_{in_3}(x) = 0$	$b_{in_3}(x) = 7x + 4$	$c_{in_3}(x) = 0$
$a_{mid_1}(x) = 7x + 4$	$b_{mid_1}(x) = 0$	$c_{mid_1}(x) = 6x + 10$
$a_{out}(x) = 0$	$b_{out}(x) = 0$	$c_{out}(x) = 7x + 4$

This gives the quadratic arithmetic program for our generalized factorization snark as

$$QAP = \{x^2+x+9, \{0, 6x+10, 0, 0, 7x+4, 0\}, \{0, 0, 6x+10, 7x+4, 0, 0\}, \{0, 0, 0, 0, 6x+10, 7x+4\}\}$$

Now as we recall, the main point for using QAPs in snarks is the fact, that solutions to R1CS are in 1:1 correspondence to the divisibility of a polynomial p, constructed from a R1CS solution and the polynomials of the QAP and the target polynomial.

So lets see this in our example. We already know from example XXX, that $I = \{1, 2, 3, 4, 6, 11\}$ *is a solution to the R1CS XXX of our problem. To see how this translates to polyinomial divisibility we compute the polynomial* p_I *by*

$$\begin{aligned}
p_I(x) = (\sum_{h\in|I|} I_h \cdot a_h(x)) \cdot (\sum_{h\in|I|} I_h \cdot b_h(x)) - (\sum_{h\in|I|} I_h \cdot c_h(x)) \\
=&(2(6x + 10) + 6(7x + 4)) \cdot (3(6x + 10) + 4(7x + 4)) - (6(6x + 10) + 11(7x + 4)) \\
=&((12x + 7) + (3x + 11)) \cdot ((5x + 4) + (2x + 3)) - ((10x + 8) + (12x + 5)) \\
=&(2x + 5) \cdot (7x + 7) - (9x) \\
=&(x^2 + 2 \cdot 7x + 5 \cdot 7x + 5 \cdot 7) - (9x) \\
=&(x^2 + x + 9x + 9) - (9x) \\
=&x^2 + x + 9
\end{aligned}$$

And as we can see in this particular example $p_I(x)$ *is equal to the target polynomial* $t(x)$ *and hence it is divisible by* t *with* $p/t = 1$.

To give a counter example we already know from XXX that $I = \{1, 2, 3, 4, 8, 2\}$ *is not a solution to our R1CS. To see how this translates to polyinomial divisibility we compute the polynomial* p_I *by*

$$\begin{aligned}p_I(x) = &(\sum_{h\in|I|} I_h \cdot a_h(x)) \cdot (\sum_{h\in|I|} I_h \cdot b_h(x)) - (\sum_{h\in|I|} I_h \cdot c_h(x))\\ =&(2(6x+10)+6(7x+4)) \cdot (3(6x+10)+4(7x+4)) - (6(6x+10)+11(7x+4))\\ =&8x^2+11x+3\end{aligned}$$

This polynomial is not divisible by the target polynomial t since Not divisible by t: $(8x^2 + 11x + 3)/(x^2 + x + 9) = 8 + \frac{3x+8}{x^2+x+9}$

5.1.5 Quadratic span programs

5.2 proof system

Now a *proof system* is nothing but a game between two parties, where one parties task is to convince the other party, that a given string over some alphabet is a statement is some agreed on language. To be more precise. Such a system is more over *zero knowledge* if this possible without revealing any information about the (parts of) that string.

Definition 5.2.0.1 ((Interactive) Proofing System)**.** *Let L be some formal language over an alphabet Σ. Then an* ***interactive proof system*** *for L is a pair* (P, V) *of two probabilistic interactive algorithms, where P is called the* ***prover*** *and V is called the* ***verifier****.*

Both algorithms are able to send messages to one another. Each algorithm only sees its own state, some shared initial state and the communication 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 in an accept state or in a reject state. We impose no restrictions on the local computation conducted by the prover.

We require that, whenever the verifier is executed the following two conditions hold:

- *(Completeness) If a string* $x \in \Sigma^*$ *is a member of language L, that is* $x \in L$ *and both prover and verifier follow the protocol; the verifier will accept.*
- *(Soundness) If a string* $x \in \Sigma^*$ *is not a member of language L, that is* $x \notin L$ *and the verifier follows the protocol; the verifier will not be convinced.*
- *(Zero-knowledge) If a string* $x \in \Sigma^*$ *is a member of language L, that is* $x \in L$ *and the prover follows the protocol; the verifier will not learn anything about x but* $x \in L$*.*

In the context of zero knowledge proving systems definition XXX gets a slight adaptation:

- Instance: Input commonly known to both prover (P) and verifier (V), and used to support the statement of what needs to be proven. This common input may either be local to the prover-verifier interaction, or public in the sense of being known by external parties (Some scientific articles use "instance" and "statement" interchangeably, but we distinguish between the two.).
- Witness: Private input to the prover. Others may or may not know something about the witness.

- Relation: Specification of relationship between instances and witness. A relation can be viewed as a set of permissible pairs (instance, witness).
- Language: Set of statements that appear as a permissible pair in the given relation.
- Statement:Defined by instance and relation. Claims the instance has a witness in the relation(which is either true or false).

The following subsections define ways to describe checking relations that are particularly useful in the context of zero knowledge proofing systems

5.2.1 Succinct NIZK

Blum, Feldman and Micali extended the notion tonon-interactivezero-knowledge(NIZK) proofs in the common reference string model. NIZK proofs are useful in theconstruction of non-interactive cryptographic schemes, e.g., digital signatures and CCA-secure public key encryption.

Definition 5.2.1.1. *Let $\mathcal{R}$ be a relation generator that given a security parameter λ in unary returns a polynomial time decidable binary relation R. For pairs $(i, w) \in R$ we call i the instance*[1] *and w the witness. We define R_λ to be the set of possible relations R the relation generator may output given 1^λ. We will in the following for notational simplicity assume λ can be deduced from the description of R. The relation generator may also output some side information, an auxiliary input z, which will be given to the adversary. An efficient prover publicly verifiable non-interactive argument for R is a quadruple of probabilistic polynomial algorithms* ($\textsc{Setup}, \textsc{Prove}, \textsc{Vfy}, \textsc{Sim}$) *such*

- *Setup: $(CRS, \tau) \rightarrow Setup(R)$: The setup produces a common reference string CRS and a simulation trapdoor τ for the relation R.*
- *Proof: $\pi \rightarrow Prove(R, CRS, i, w)$: The prover algorithm takes as input a common reference string CRS and a statement $(i, w) \in R$ and returns an argument π.*
- *Verify: $0/1 \rightarrow Vfy(R, CRS, i, \pi)$: The verification algorithm takes as input a common reference string CRS, an instance i and an argument π and returns 0 (reject) or 1 (accept).*
- *$\pi \rightarrow Sim(R, \tau, i)$: The simulator takes as input a simulation trapdoor τ and instance i and returns an argument π.*

Groth16

Groth's constant size NIZK argument is based on constructing a set of polynomial equations and using pairings to efficiently verify these equations. Gennaro, Gentry,Parno and Raykova [Pinocchio] found an insightful construction of polynomial equations based on Lagrange interpolation polynomials yielding a pairing-based NIZK argumentwith a common reference string size proportional to the size of the statement and wit-ness.

It constructs a snark for arithmetic circuit satisfiability, where a proof consists of only 3 group elements. In addition to being small, the proof is also easy to verify. The verifier

[1]Note that in Groth16 this is called the statement. We think the term instance is more consistent with SOMETHING.

just needs to compute a number of exponentiations proportional to the instance size and check a single pairing product equation, which only has 3 pairings.

The construction can be instantiated with any type of pairings including Type III pairings, which are the most efficient pairings. The argument has perfect completeness and perfect zero-knowledge. For soundness ??

In the common reference string model.

Example 37 (Generalized factorization snark)**.** *In this example we want to compile our main example in Groth16. Input is the R1CS from example 35. We choose the following parameters*

$$\textit{curve} = \textit{BLS6-6} \quad \mathbb{G}_1 = \textit{BLS6-6(13)} \quad g_1 = (13,15) \quad \mathbb{G}_2 = \quad g_2 =$$

Setup phase:

Our pipeline starts by computing the polynomial

Chapter 6

Exercises and Solutions

TODO: All exercises we provided should have a solution, which we give here in all detail.