

# Moonmath manual

TechnoBob and the Least Scruples crew

July 14, 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. They allow proving the veracity of a statement, related to confidential data, without revealing any information beyond the validity of the statement. ZKPs were initially developed by the academic community in the 1980s, and have seen tremendous improvements since then. They are now of practical feasibility in multiple domains of interest to the industry, and to a large community of developers and researchers. ZKPs can have a positive impact in industries, agencies, and for personal use, by allowing privacy-preserving applications where designated private data can be made useful 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 to convince the verifier that a statement is true, without revealing any additional information. For example, suppose the prover holds a birth certificate digitally signed by an authority. In order to access some service, the prover may have to prove being at least 18 years old, that is, that there exists a birth certificate, tied to the identity of the prover and digitally signed by a trusted certification authority, stating a birthdate consistent with the age claim. A ZKP allows this, without the 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 (2017): 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](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, for 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 cryptographic 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 Hash functions

We assume that  $H : \{0,1\}^* \rightarrow 0,1^k$  is a **hash function** that maps binary strings of arbitrary length onto strings of length  $k$ . In addition we define a hash function to be  $l$ -bounded if it is only able to map from binary strings of length  $l$  to binary strings of length  $k$ .

### STUFF ON CRYPTOGRAPHIC HASH FUNCTIOND

**p&p-hash** In this example we define a 16-bounded pen&paper hash function that is simple enough to be computed without a computer. We call it the PaP-Hash and will use it throughout the book as a basic example whenever hashing is involved in other example.

The PaP-Hash  $\mathcal{H}_{PaP} : \{0,1\}^{16} \rightarrow \{0,1\}^4$  is defined in the following way:

- Decompose the 16-bit preimage  $S = (s_0, s_1, \dots, s_{15})$  into 4 chunks  $S_i = (s_{4i+0}, \dots, s_{4i+3})$  for  $i \in \{0, 1, 2, 3\}$ .
- For each chunk  $S_i$  do a circular bitshift  $s_j \rightarrow s_{j+1} \bmod 4$  for all  $s_j \in S_i$
- Xor all four chunks together  $S = S_1 \text{ XOR } S_2 \text{ XOR } S_3 \text{ XOR } S_0$
- Compute the result  $\mathcal{H}_{PaP}(S) = S \text{ XOR } (1001)$

**Example 1.** Lets compute our PaP-Hash on a concrete example string  $S = (1110011101110011)$ . Then the decomposition is  $S_0 = (1110)$ ,  $S_1 = (0111)$ ,  $S_2 = (0111)$  and  $S_3 = (0011)$  and after a circular bitshift we get  $S'_0 = (0111)$ ,  $S'_1 = (1011)$ ,  $S'_2 = (1011)$  and  $S'_3 = (1001)$ . Xoring everything together we get  $S = (0111) \text{ XOR } (1011) \text{ XOR } (1011) \text{ XOR } (1001) = (1100) \text{ XOR } (0010) = (1110)$ . So we get  $\mathcal{H}_{PaP}(10100111011100011) = (1110)$ .

## 3.5 Software Used in This Book

In 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 website<sup>1</sup>. 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

---

<sup>1</sup><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 encouraged to do some exercises from the appendix 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 rational 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 them abstract we arrive at the definition of a group:

**Definition 4.1.0.1** (Group). A group  $(\mathbb{G}, \cdot)$  is a set  $\mathbb{G}$ , together with a map  $\cdot : \mathbb{G} \cdot \mathbb{G} \rightarrow \mathbb{G}$ , 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 *Magnas*, 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() 1
Category of groups 2
sage: CommutativeAdditiveGroups() 3
Category of commutative additive groups 4
sage: FiniteGroups() 5
Category of finite groups 6
```

**Example 2** (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 3** (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) 7
```

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  $+: R \cdot R \rightarrow R$  and  $\cdot: R \cdot R \rightarrow R$ , 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() 8
Category of commutative rings 9
sage: CommutativeRings().super_categories() 10
[Category of rings, Category of commutative monoids] 11
```

**Example 4** (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 12
Integer Ring 13
sage: ZZ(5) # Get an element from the Ring of integers 14
5 15
sage: ZZ(5) + ZZ(3) 16
8 17
sage: ZZ(5) * ZZ(3) 18
15 19
sage: ZZ.random_element(10**50) 20
3154069053411604875544738921854054222706186002740 21
sage: ZZ(27713).str(2) # Binary string representation 22
110110001000001 23
sage: ZZ(27713).str(16) # Hexadecimal string representation 24
6c41 25
```

**Example 5** (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 6.** Let  $S := \{\bullet, \star, \odot, \otimes\}$  be a set that contains four elements and let addition and multiplication on  $S$  be defined as follows:

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

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

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

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

To see how such 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"

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

On the other hand, EQUATION  $2x = 3$  has no solution and equation BLA has more than 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  $+: \mathbb{F} \cdot \mathbb{F} \rightarrow \mathbb{F}$  and  $\cdot: \mathbb{F} \cdot \mathbb{F} \rightarrow \mathbb{F}$ , 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:

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

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 than the neutral element of addition has an inverse.

```
sage: Fields() 26
Category of fields 27
```

**Example 7** (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 28
Rational Field 29
sage: QQ(1/5) # Get an element from the field of rational 30
          numbers
1/5 31
sage: QQ(1/5) / QQ(3) # Division 32
1/15 33
```

**Example 8** (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:

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \quad \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

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) 34
Finite Field of size 2 35
sage: GF(2)(1) # Get an element from GF(2) 36
1 37
sage: GF(2)(1) + GF(2)(1) # Addition 38
0 39
sage: GF(2)(1) / GF(2)(1) # Division 40
1 41
```

### 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 division 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*

$$a = m \cdot b + r \quad (4.1)$$

*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  $a \text{ div } b := m$  to describe the quotient and use the symbol  $a \bmod b := r$  for the remainder. We also say, that an integer  $a$  is divided by a number  $b$  if  $a \bmod b = 0$  holds. In this case we also write  $a|b$  (? Note 1 below).*

**Example 9.** *Using our previously defined notation, we have  $-17 \text{ div } 4 = -5$  and  $-17 \bmod 4 = 3$ , 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 remainder is not zero. Writing  $-17|4$  therefore has no meaning.*

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

<code>sage: ZZ(-17) // ZZ(4) # Integer quotient</code>	42
<code>-5</code>	43
<code>sage: ZZ(-17) % ZZ(4) # remainder</code>	44
<code>3</code>	45
<code>sage: ZZ(-17).divides(ZZ(4))</code>	46
<code>False</code>	47
<code>sage: ZZ(4).divides(ZZ(12))</code>	48
<code>True</code>	49

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 10** (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 remainder  $m, r \in \mathbb{N}$  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 50
(604765, 188) 51
sage: ZZ(604765)*ZZ(261) + ZZ(188) # check 52
157843853 53
```

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

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

holds. Two numbers are called **relative prime**, if their greatest 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
     $q_k := r_{k-1} \text{ div } r_k$ 
     $r_{k+1} := r_{k-1} - q_k \cdot r_k$ 
     $s_{k+1} := s_{k-1} - q_k \cdot s_k$ 
     $k \leftarrow k + 1$ 
end while
```

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

**Example 11.** *To illustrate the algorithm, lets apply it to the numbers (12, 5). We compute*

$k$	$r_k$	$s_k$	$t_k = (r_k - s_k \cdot a) \text{ div } b$
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) 54
(1, -2, 5) 55
```

### 4.1.2 Modular arithmetic

Congruence or modular 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*

$$a \bmod n = b \bmod n \quad (4.3)$$

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

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

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

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

56  
57

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:*

$$k^p \equiv k \pmod{p}, \quad (4.4)$$

*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
```

58  
59  
60  
61

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*

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\dots \\ x &\equiv a_k \pmod{n_k} \end{aligned} \tag{4.5}$$

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

*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  $N = n_1 \cdot n_2 \cdot \dots \cdot n_k$
- For each  $1 \leq j \leq k$ , compute  $N_j = \frac{N}{n_j}$
- For each  $1 \leq j \leq k$ , 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 13.** *To illustrate how to solve simultaneous congruences using the Chinese remainder theorem, let's look at the following system of congruencies:*

$$\begin{aligned} x &\equiv 4 \pmod{7} \\ x &\equiv 1 \pmod{3} \\ x &\equiv 3 \pmod{5} \\ x &\equiv 0 \pmod{11} \end{aligned}$$

*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*

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

*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])`

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  $n$ . It 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 first equivalence class and an 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 precise

**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 14** (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*

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

*Now to compute the addition of those equivalence classes, say  $2 + 5$ , one chooses arbitrary elements from both sets say 14 and  $-1$ , adds those numbers in the usual way and then looks 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  $\mathbb{Z}_6$  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

$\cdot$	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      64
sage: Z6(2)+Z6(5) # standard representatives of a class 65
1                                                    66
sage: Z6(14)+Z6(-1) # different representatives for same 67
class
1                                                    68
sage: - Z6(2) # additive inverse                    69
4                                                    70
sage: Z6(5)**(-1) # multiplicative inverse if exists 71
5                                                    72
```

TODO:

Barrett reduction

Montgomery modular multiplication (Montgomery domain)

### 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*

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

*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 15.** *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']      73
sage: Z6x                74
Univariate Polynomial Ring in x over Ring of integers modulo 75
6
sage: p = Z6x([1,2,3,4]) 76
sage: p                  77
```

```
4*x^3 + 3*x^2 + 2*x + 1
```

78

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

```
sage: p.degree()
```

79

3

80

```
sage: Z6x([0]).degree()
```

81

-1

82

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

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

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

```
sage: q = Z6x([5, -3, 2,])
```

83

```
sage: p + q
```

84

```
4*x^3 + 5*x^2 + 5*x
```

85

```
sage: p*q
```

86

```
2*x^5 + 3*x^3 + 5*x^2 + x + 5
```

87

```
sage: p^2
```

88

```
4*x^6 + x^4 + 2*x^3 + 4*x^2 + 4*x + 1
```

89

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 connected to 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

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

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:*

$$n = p_1 \cdot p_2 \cdot \dots \cdot p_k . \quad (4.10)$$

*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 that 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 cryptosystems, 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 to 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 division.

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 nothing 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 let's 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 16.** To summarize the basic aspects of computation in prime fields, let's consider the prime field  $\mathbb{F}_5$  and simplify the following expression

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

A first thing to note is that since  $\mathbb{F}_5$  is a field all rules like bracketing (distributivity), summing etc. are identical to the rules we learned in school when we were 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  $(\text{mod } 4, 5) = 4$  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 fraction, we need the multiplicative inverse of 3, which is the number, that when multiplied 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 computations of each  $t_k$  is irrelevant here:

$k$	$r_k$	$x_k^{-1}$	$t_k = (r_k - s_k \cdot a) \operatorname{div} b$
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  $\frac{4}{3} + 1 = 4 \cdot 2 + 1 = 3 + 1 = 4$ .

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*

$$x^p = x \quad \text{or} \quad x^{p-1} = 1. \quad (4.11)$$

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

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

Addition and multiplication can 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

$\cdot$	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  $4 \cdot 4 = 1$ .

**Hash to Prime fields** An important problem in elliptic curve cryptography and in its implementations as a snark is the ability to hash to (various subsets) of elliptic curves. As we will see in XXX those curves are usually defined over prime fields and hashing to a curve often starts with hashing to the prime field. In this paragraph we therefore look at common techniques to hash to a prime field.

In what follows let  $\mathbb{F}_q$  be a prime field, such that  $q$  is a prime number with  $m$ -digits in its binary representation, i.e.  $|p_{base2}| = m$  and let  $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$  be a hash function. The methods to map  $H$  onto  $\mathbb{F}_q$  depend on  $k$ .

If  $k \leq m - 1$ , then every image  $H(data)$  if interpreted as an integer in its base-2 representation, is smaller than  $p$  and hence can directly be interpreted as an element of  $\mathbb{F}_q$ . So in this case  $H : \{0, 1\}^* \rightarrow \mathbb{F}_q$  can be used unchanged. The drawback of this simple method, is that the bigger the difference between  $k$  and  $m$  is, the more will the distribution of  $H$  deviate from uniformity.

For example in the extreme case  $k = 1$ ,  $H$  only maps to  $\{0, 1\} \subset \mathbb{F}_q$ . The best possible case is therefore  $k = m - 1$ . In that case only the highest XXX numbers (DO THE COMPUTATION) are missing from the distribution.

On the other hand if  $k \geq m$ , then there are basically two commonly used methods to map the output of  $H$  onto  $\mathbb{F}_q$ . The first is to simply forget all leading  $k - m + 1$ -bits from the image of  $H$ , which brings you back to our previous consideration.

The second method is to interpret  $H(data)$  as an integer and then compute the modulus  $H(data) \bmod p$ . This also introduces a small bias (COMPUTATION FROM THE FORUM ENTRY).

**Example 18** (p&p- $\mathbb{F}_{13}$ -mod-hash). *Consider our pen&paper hash function from XXX. We want to use this hash function, to define a 16-bounded hash function that maps into the prime field  $\mathbb{F}_{13}$ . We define:*

$$\mathcal{H}_{mod}^{13} : \{0, 1\}^{16} \rightarrow \mathbb{F}_{13} : S \mapsto \mathcal{H}_{PaP}(S) \bmod 13$$

*Considering the string  $S = (1110011101110011)$  from example XXX again we know  $\mathcal{H}_{PaP}(S) = (1110)$  and since  $(1110)_{10} = 14$  and  $14 \bmod 13 = 1$  we get  $\mathcal{H}_{mod}^{13}(S) = 1$ .*

**Example 19** (p&p- $\mathbb{F}_{13}$ -drop-hash). *We can consider the same pen&paper hash function from XXX and define another hash into  $\mathbb{F}_{13}$ , by deleting the first leading bit from the hash. The result is then a 3-digit number and therefore guaranteed to be smaller than 13, since 13 is equal to  $(1101)$  in base 2.*

*Considering the string  $S = (1110011101110011)$  from example XXX again we know  $\mathcal{H}_{PaP}(S) = (1110)$  and stripping of the leading bit we get  $(110)_{10} = 6$  as our hash value.*

*As we can see this hash function has the drawback of an uneven distribution in  $\mathbb{F}_{13}$ . In fact this hash function is unable to map to values from  $\{8, 9, 10, 11, 12\}$  as those numbers have a 1-bit in position 4. However as we will see in XXX, this hash is cheaper to implement as a circuit as no expensive modulus operation has to be used.*

**Definition 4.2.0.4** (The finite field). *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

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

for the set of all square roots of  $y$  in the prime field  $\mathbb{F}_p$ . (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 20** (square numbers and roots in  $\mathbb{F}_5$ ). Let us consider our example (17) 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:

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

**Example 21.** If we look again at the example (17) we have the following Legendre symbols

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

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

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

*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 Tonelli ? 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  $t^{2^i} = 1$   
      $b := c^{2^{M-i-1}}$   
      $M := i, \quad c := b^2, \quad t := tb^2, \quad R := R \cdot b$   
**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

Elliptic 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

$$\mathbb{F}_{p^m} := \{a_{k-1}t^{k-1} + a_{k-2}t^{k-2} + \dots + a_1t + a_0 \mid a_i \in \mathbb{F}_p\}$$

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 22** (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 irreducible 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 polynomials of degrees lower than 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	t+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	t+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  $\mathbb{F}_{32}$ . For example in  $\mathbb{F}_{32}$  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.

As we can see from the previous example, it can become quite messy to write elements of extension fields. Especially for larger extension degrees. In the literature we therefore find two ways to have a nicer description.

The most obvious one is to observe that polynomials of maximal degree  $m$  are in one to one correspondence with  $m + 1$ -dimensional vectors by

$$\sum_{j=0}^m a_j x^j \Leftrightarrow (a_0, a_1, \dots, a_m)$$

so the coordinate vector associated to a given polynomial is given by its coefficients and vice versa. This way  $\mathbb{F}_{q^m}$  can be seen as an  $m$ -dimensional vector space over the prime field  $\mathbb{F}_q$ . Addition is the usual component wise addition of vectors. However multiplication is not obvious and must be computed from the polynomial OR SUBSTITUTION.

**Example 23.** In this example the set  $\mathbb{F}_{32}$  can be written as  $\{[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1]\}$  and we get for example  $[1, 1] + [2, 1] = [0, 2]$  by adding component wise.

Power representation

## 4.3 Epileptic Curves

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

The elliptic curves we consider are all defined over Galois 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  $E/\mathbb{F}_q$  over  $\mathbb{F}_q$  is defined as the set

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

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 all 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 size of an underlying field.

**Example 24.** *Lets consider our prime field  $\mathbb{F}_5$  from (XXX). If we choose  $a = 1$  and  $b = 0$  then  $4a^3 + 27b^2 = 4 \neq 0$  and the corresponding elliptic curve  $E/\mathbb{F}_5$  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. Sage code for this curve:*

<code>sage: EllipticCurve(GF(5), [1, 0])</code>	90
<code>Elliptic Curve defined by y^2 = x^3 + x over Finite Field of</code>	91
<code>size 5</code>	
<code>sage: EllipticCurve(GF(5), [1, 0]).trace_of_frobenius()</code>	92
<code>2</code>	93

**Example 25.** *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  $-3$ .*

**Keys** - Keys on elliptic curves - compressed keys

## 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  $P \neq \mathcal{O}$  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 elliptic 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-tangent-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

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

- (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*

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

where  $[m]P = P + P + \dots + P$  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  $x \cdot x \cdot \dots \cdot x$  is usually written in as  $x^m$ . So our exponential map, is an adaption of that for groups with an additive notation of the group law.*

**Example 26.** *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

$$\begin{aligned}(0, 0) + (2, 0) &= (3, 0) \\ (0, 0) + (3, 0) &= (2, 0) \\ (2, 0) + (3, 0) &= (0, 0)\end{aligned}$$

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 27.** 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 exist. On the other hand if one of the elements is a generator, then infinite many solutions exist.

## 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

### 4.4.1 twists

## 4.5 Pairings

In this section, we discuss *pairings*, which form the basis of several zk-SNARKs and other zero knowledge proof schemes. The SNARKs derived from pairings have the advantage of constant-sized proof sizes, which is crucial to blockchains.

We start out by defining pairings and discussing a simple application which bears some resemblance to the more advanced SNARKs. We then introduce the pairings arising from elliptic curves and describe Miller's algorithm which makes these pairings practical rather than just theoretically interesting.

### 4.5.1 Hashing to Curve

In various cryptographic primitives the ability to generate elliptic curve points from binary strings that have properties similar to those of cryptographically secure hash functions is desirable. We call this hash-to-curve.

Of particular interest are constructions, that not only hash to curve points, but where the images have special properties, like being in the pairing group  $\mathbb{G}_1$  or  $\mathbb{G}_2$ .

Thinking about hashing to curves and hashing to certain subgroups in particular maybe the most obvious thing that comes to mind, is to simply hash to the scalar field of the elliptic curve and then use a generator of the subgroup target and multiply the generator with the hash value in that scalar field. The result is a valid curve point that is guaranteed to be in the desired subgroup.

To be more precise given an elliptic curve  $E/\mathbb{F}$ , a subgroup  $\mathbb{G}$  of  $E$  of order  $r$ , a generator  $g$  of  $\mathbb{G}$ , a bit-string  $s$  and a hash function  $H : \{0, 1\}^* \rightarrow \mathbb{F}_r$ , then  $[H(s)]g$  is an element of  $\mathbb{G}$  and hence  $H' : \{0, 1\}^* \rightarrow \mathbb{G}$  with  $H'(s) := [H(s)]g$  is a hash function.

This naive approach however is almost never adequate in cryptographic applications as a discrete log relation is always known between any two given hash value  $H'(s)$  and  $H'(t)$ . In fact for  $H(s) \neq 0$ , we can define  $x = H(t)/H(s)$  and then have  $[x]H'(s) = H'(t)$ .

So we need a different approach:

**Try and increment hash functions** One of the most straight forward ways to hash a bitstring onto an elliptic curve point, in a secure way, is to use a cryptographic hash function together with one of the methods we described in XXX to hash to the base field of the curve. Ideally the hash function generates an image that is at least one bit longer than the bit representation of the base field modulus.

The image in the base field can then be interpreted as the  $x$ -coordinate of the curve point and the two possible  $y$ -coordinates are then derived from the curve equation, while one of the bits that exceeded the modulus determines which of the two  $y$ -coordinates to choose.

Such an approach would be easy to implement and deterministic and it will conserve the cryptographic properties of the original hash function. However not all  $x$ -coordinates generated in such a way, will result in quadratic residues, when inserted into the defining equation. It follows that not all field elements give rise to actual curve points. In fact on a prime field, only half of the field elements are quadratic residues and hence assuming an even distribution of the hash values in the field, this method would fail to generate a curve point in about half of the attempts.

One way to account for this problem is the so called *try and increment* method. Its basic assumption is, that hashing different values, the result will eventually lead to a valid curve point.

Therefore instead of simply hashing a string  $s$  to the field the concatenation of  $s$  with additional bytes is hashed to the field instead. The bytes are initially zero and interpreted as an unsigned integer. If the first *try* of hashing to the field does not result in a valid curve point, the counter is *incremented* and hashed again. This is repeated until a valid curve point is found eventually.

This method has the advantage that is relatively easy to implement in code and that it preserves the cryptographic properties of the original hash function. However it is not guaranteed to find a valid curve point, as there is a chance that all possible values in the chosen size of the counter bytes fail to generate a quadratic residue. Fortunately it is possible to make the probability for this arbitrarily small by choosing large enough counters and relying on the (approximate) uniformity of the hash-to-field function.

One might think that another disadvantage of this method in the context of snarks is that it can not be implemented as a circuit effectively. This however is not fully true, as a circuit/r1cs only needs to enforce the correctness of the computation. Hence for the circuit it is enough to check the hash of the string and the correct counter. It does not need to find that counter.

So to be more formal, we can define a try and increment hash-to-curve like this

DEF

Considering certain subgroups of the elliptic curve, the usefulness of this methods depends highly on the actual situation. For example if a hash to the  $n$ -torsion subgroup  $\mathbb{G}_1$  is desired, there are two possibilities:

First generic try and increment can be used, followed by a cofactor clearing step. This way every hash on the curve is considered valid, but then projected to  $\mathbb{G}_1$  afterwards

Second, the try step not only checks if the curve point actually exists, but also if it is a point in  $\mathbb{G}_1$ . If its not then the increment step is executed until a valid point is found. This is possible in most applications, since  $\mathbb{G}_1$  is usually the by far largest subgroup in  $E$  and the probability to find a point in it is large.

The situation for  $\mathbb{G}_2$  as defined in XXX is different and the try and increment method usually fails to find hash values in  $\mathbb{G}_2$ . For once  $\mathbb{G}_2$  is defined in  $E$  over an extension field, not the prime field itself, so hashing to  $\mathbb{F}$  must be extended into a hash to the extension field. This is possible, but not desirable eventually, because even if we find valid curve points in the curve,  $\mathbb{G}_2$ . We therefore need different approaches for hashing into  $\mathbb{G}_2$

## Pederson Hashes

**Definition 4.5.1.1** (Pedersen's Hash function). *Let  $\mathbb{G}$  be a cyclic group of order  $r$  and  $\{g_1, \dots, g_k\} \subset \mathbb{G}$  a uniform randomly generated set of generators of  $\mathbb{G}$ . Then **Pedersen's hash function***

$$H : (\mathbb{F}_r)^k \rightarrow \mathbb{G}$$

*is defined for any message  $M = (M_1, \dots, M_k) \in \mathbb{F}_r \times \dots \times \mathbb{F}_r$ , by  $H(M) = \prod_{j=1}^k [M_j]g_j$ .*

### SIMPLE WORD DESCRIPTION

**Remark 17.** *Of course Pedersen's hash function can be combined with a hash-to-field function as described in XXX to give a function that maps ordinary bit strings to group*

points. However for this function to be secure, it is necessary to prove that there is no known discrete log relations between the elements of the generator set.

It can be shown that collision resistance is equivalent to the hardness of the discrete log problem XXX.

**Example 28.**

Posaidon Hashes

## 4.5.2 Special Functions

**Theorem 4.5.2.1** (Pseudo-Randomness in cyclic groups). *Let  $\mathbb{G}$  be a cyclic group of order  $r$ , such that DDT is hard in  $\mathbb{G}$ , let  $g$  be a generator and  $\{a_0, a_1, \dots, a_n\} \subset \mathbb{F}_r^*$  a uniform randomly generated set of invertible field elements, from the scalar field of  $\mathbb{G}$ . Then the function*

$$H : \{0, 1\}^n \rightarrow \mathbb{G}$$

*defined for any binary string  $s = (s_1, \dots, s_n)$  by  $H(s) = [a_0 \cdot \prod_{j=1}^n a_j^{s_j}]g$  is a pseudo-random function.*

## 4.6 Constructing Elliptic curves

### 4.6.1 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  $CL(Dv^2) = \{(a, b, c) \mid a, b, c \in \mathbb{Z}, |b| \leq a \leq \sqrt{\frac{Dv^2}{3}}, a \leq c, b^2 - 4ac = -Dv^2, (a, b, c) = 1\}$ . If  $|b| = a$  or  $a = c$ , then  $b \geq 0$ .
- Compute  $H_D(x) = \prod_{(a,b,c) \in CL(D)} (x - j(\frac{-b + \sqrt{-Dv^2}}{2a}))$
- Round the coefficients of  $H_D$  to the closed integers.
- Compute  $H_{D,p} = H_D \bmod_p$
- 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  $k = \frac{j}{1728-j}$ .

## 4.7 Classes of elliptic curves

In this section we describes ways to describe elliptic curves different from the general Weierstrass form. Alternative descriptions are sometimes useful because of DIFFERENT ways to express the group laws

### 4.7.1 Montgomery Form

Let  $\mathbb{F}$  be a finite field with characteristic  $\neq 2$ . A Montgomery curve over  $\mathbb{F}$  is defined by the equation

$$M : By^2 = x^3 + Ax^2 + x$$

for certain  $A, B \in \mathbb{F}$ , with  $A \neq \pm 2$ ,  $b \neq 0$  and  $B(A^2 - 4) \neq 0$ .

The Montgomery form is just another form to define an elliptic curve. In fact every curve in Montgomery form can be transformed into an elliptic curve in Weierstrass form and vice versa. To see that assume that a curve in Montgomery form  $By^2 = x^3 + Ax^2 + x$  is given. The associated Weierstrass form is then

$$y^2 = x^3 + \frac{3 - A^2}{3B^2} \cdot x + \frac{2A^3 - 9A}{27B^3}$$

On the other hand, an elliptic curve over base field  $\mathbb{F}$  in Weierstrass form  $E : v^2 = t^3 + at + b$  can be converted to Montgomery form if and only if the following conditions hold:

- $E$  has order divisible by 4
- The polynomial  $z^3 + az + b \in \mathbb{F}[]$  has at least one root  $\alpha \in \mathbb{F}$
- $3\alpha^2 + a$  is a quadratic residue in  $\mathbb{F}$ .

When these conditions are satisfied, then for  $s = (\sqrt{3\alpha^2 + a})^{-1}$  the equivalent Montgomery curve is given by

$$sy^2 = x^3 + (3\alpha s)x^2 + x$$

### 4.7.2 Twisted Edwards Form

In this section we describe curves, that are defined over Galois fields of characteristics  $\neq 2$ . A **twisted Edwards curve** over a Galois field  $\mathbb{F}$  is a curve, defined by the equation

$$E/\mathbb{F} : a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$$

where  $x, y \in \mathbb{F}$  and  $d \in \mathbb{F} \setminus \{0, 1\}$  and  $d \in \mathbb{F} \setminus \{1\}$  with  $a \neq d$ . Such a curve is called an Edwards curve (non twisted), if  $a = 1$ .

One of our main goals is to implement example snarks defined over the scalar field of our BLS6 curve. In particular we want to do elliptic curve cryptography inside of circuits. To do so we need an elliptic curve that we can implement as a circuit over the scalar field of BLS6.

As we have seen in XXX Weierstrass curve have somewhat complicated addition and doubling laws as many cases have to be distinguished. Those cases translate to branches in computer programs, which are costly when implemented in circuits/r1cs. It is therefore advantageous to look for curves with a more simple addition/doubling rule.

Edwards curves are particular useful here as they have a compact addition law that works for all point. Implementing that rule therefore needs no branching.

The most remarkable fact about twisted Edwards curves is their simple addition law. Assuming that  $a$  has a root in  $\mathbb{F}$  and  $d$  has no root in  $\mathbb{F}$ , the sum of any two points  $(x_1, y_1)$ ,  $(x_2, y_2)$  on such an Edwards curve  $E$  is given by

$$(x_1, y_1) + (x_2, y_2) = \left( \frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right)$$

The point  $(0, 1)$  is the neutral element of the addition law. The inverse of a point  $(x_1, y_1)$  on  $E$  is  $(-x_1, y_1)$ . The addition law is very simple to implement and it can also be used for point doubling. It also works for the neutral element, for inverses..

As Edwards curves have such simple non branching addition laws that work for all points, including the neutral element and inverses, they are a interesting for snarks as they can be implemented with only a few constraints in circuits.

$(0,1)$

The twisted Edwards form is just another form to define an elliptic curve. In fact every curve in twisted Edwards form can be transformed into an elliptic curve in Montgomery form and vice versa. To see that assume that a curve in twisted Edwards form  $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$  is given. The associated Montgomery form is then

$$\frac{4}{a-d} y^2 = x^3 + \frac{2(a+d)}{a-d} \cdot x^2 + x$$

On the other hand a Montgomery curve  $By^2 = x^3 + Ax^2 + x$  (WITH STUFF) can be transformed into a twisted Edwards curve

$$\left(\frac{A+2}{B}\right)x^2 + y^2 = 1 + \left(\frac{A-2}{B}\right)x^2 y^2$$

## 4.8 Pend and Paper example curves

### 4.8.1 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  $\Phi_6$  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:

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

Since  $q(1) = 1$  is not a prime number, the first  $x$  that gives a proper curve is  $x = 2$ . However such a curve would be defined over a base field of characteristic 3 and we would rather like to avoid that. We therefore 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.

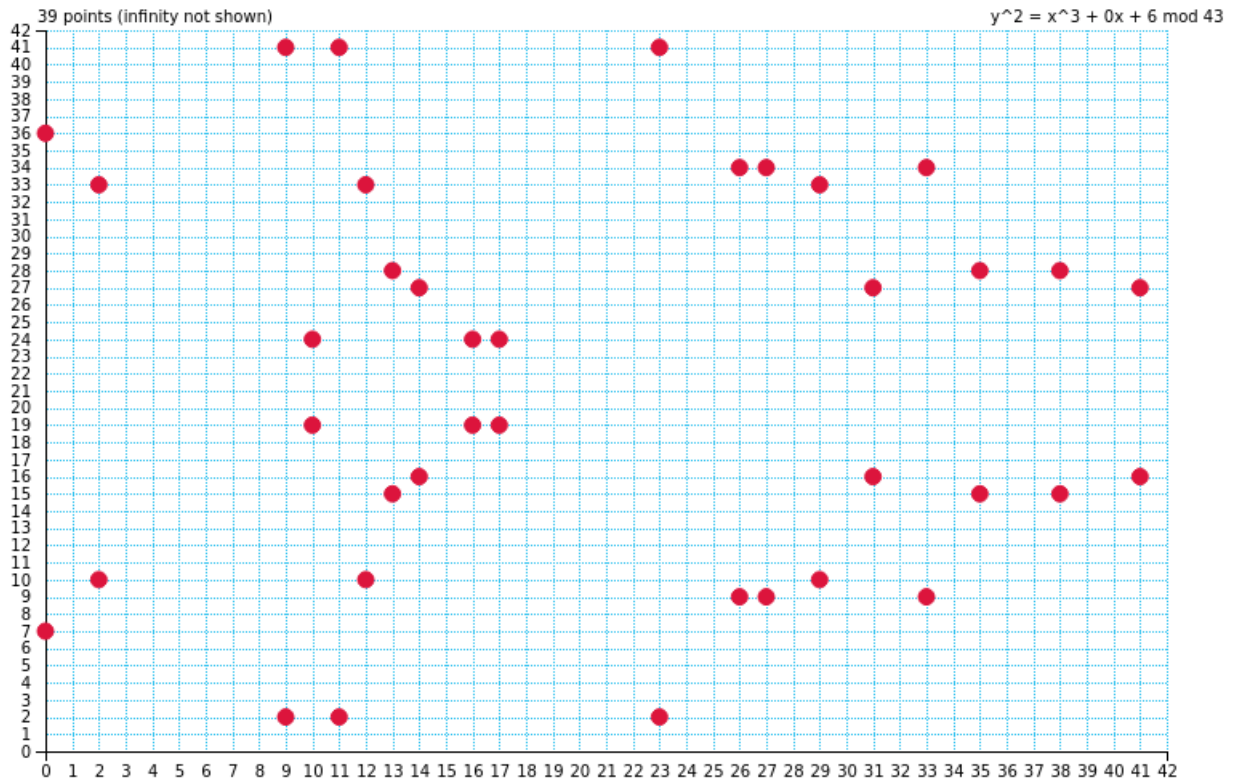
To compute the defining equation  $y^2 = x^3 + ax + b$  of *BLS6-6*, we use the complex multiplication algorithm as described above in XXX. The goal is to find  $a, b \in \mathbb{F}_{43}$  representations, that are particulary nice to work with. As shown for example in XXX the discriminant  $D$  of all 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  $j(\frac{-1+i\sqrt{3}}{2}) = 0$ . 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

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

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 print 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

$$BLS6 - 6(13) = \{(13, 15) \rightarrow (33, 34) \rightarrow (38, 15) \rightarrow (35, 28) \rightarrow (26, 34) \rightarrow (27, 34) \rightarrow (27, 9) \rightarrow (26, 9) \rightarrow (35, 15) \rightarrow (38, 28) \rightarrow (33, 9) \rightarrow (13, 28) \rightarrow \mathcal{O}\} \quad (4.16)$$

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

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

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:

	$\mathcal{O}$	(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)	$\mathcal{O}$	(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)	$\mathcal{O}$	(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)	$\mathcal{O}$	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	$\mathcal{O}$	(13, 15)
(35, 28)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	$\mathcal{O}$	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	$\mathcal{O}$	(13, 15)	(33, 34)
(26, 34)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	$\mathcal{O}$	(38, 28)	(33, 9)	(13, 28)	$\mathcal{O}$	(13, 15)	(33, 34)	(38, 15)
(27, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	$\mathcal{O}$	(13, 28)	(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  $BLS6 - 6(13)$ .

In what follows we want to compute type 2 pairings on our BLS6 curve. We therefore need to extract the subgroup  $\mathbb{G}_1$  as well as  $\mathbb{G}_2$  from the full 13-torsion group. We already know from XXX that  $\mathbb{G}_1$  is given by

$$\mathbb{G}_1 = \{(13, 15) \rightarrow (33, 34) \rightarrow (38, 15) \rightarrow (35, 28) \rightarrow (26, 34) \rightarrow (27, 34) \rightarrow (27, 9) \rightarrow (26, 9) \rightarrow (35, 15) \rightarrow (38, 28) \rightarrow (33, 9) \rightarrow (13, 28) \rightarrow \mathcal{O}\}$$

In type 2 pairings, the group  $\mathbb{G}_2$  is defined by those elements  $P$  of the full 13-torsion group, that are mapped to  $43 \cdot P$  under the Frobenius endomorphism XXX. Since  $BLS6/\mathbb{F}_{13^6}$  contains 6321251664 elements, we can not simply loop through all elements, to find the full 13-torsion group and extract all elements from  $\mathbb{G}_2$ . However we can derive the full 13-torsion as the set of all 13-division points and then extract  $G_2$  from this

```

sage: F43 = GF(43)
sage: F43t.<t> = F43[]
sage: F43_6.<v> = GF(43^6, name='v', modulus=t^6+6) # t^6+6
         irreducible
sage: BLS6 = EllipticCurve(F43_6,[0,6])
sage: INF = BLS6(0) # point at infinity
sage: for P in INF.division_points(13): # PI(P) == [q]P
.....:     if P.order() == 13: # exclude point at infinity
.....:         PiP = BLS6([a.frobenius() for a in P])
.....:         qP = 43*P
.....:         if PiP == qP:
.....:             print(P.xy())

```

Choose  $g_2 = (7v^2, 16v^3)$  as generator of  $\mathbb{G}_2$ , we get

$$\begin{aligned} \mathbb{G}_2 = \{ & (7v^2, 16v^3) \rightarrow (10v^2, 28v^3) \rightarrow (42v^2, 16v^3) \rightarrow (37v^2, 27v^3) \rightarrow \\ & (16v^2, 28v^3) \rightarrow (17v^2, 28v^3) \rightarrow (17v^2, 15v^3) \rightarrow (16v^2, 15v^3) \rightarrow \\ & (37v^2, 16v^3) \rightarrow (42v^2, 27v^3) \rightarrow (10v^2, 15v^3) \rightarrow (7v^2, 27v^3) \rightarrow \mathcal{O} \} \end{aligned}$$

e.g.  $[3]g_2 = (42v^2, 16v^3)$ .

Having those groups we can do pairings. We choose the Weil pairing and invoke `sagemath`. For example the Weil pairing between our two generators is

$$e(g_1, g_2) = 5v^5 + 16v^4 + 16v^3 + 15v^2 + 3v + 41$$

```
sage: g1 = BLS6([13, 15]) 105
sage: g2 = BLS6([7*v^2, 16*v^3]) 106
sage: g1.weil_pairing(g2, 13) 107
5*v^5 + 16*v^4 + 16*v^3 + 15*v^2 + 3*v + 41 108
```

As we have seen,  $\mathbb{G}_2$  needs quite a bit more storage space than  $\mathbb{G}_1$ , since elements in  $\mathbb{G}_2$  are pairs of polynomials of degree  $< 6$  with coefficients in  $\mathbb{F}_{43}$ , while elements from  $\mathbb{G}_1$  are just pairs of elements from  $\mathbb{F}_{43}$ .

As we know from XXX it is possible to reduce the space needed to store  $\mathbb{G}_2$  by using the concept of a twist. In our case *BLS6* has embedding degree 6 and the curve parameter  $a$  in  $y^2 = x^3 + ax + b$  is zero. We therefore know from XXX, that *BLS6* has three different twist: A quadratic twist, a cubic twist and a sextic twist. We want to compute all of these twist:

The quadratic twisted *BLS6-6* curve: Consider our *BLS6-6* curve  $BLS6 - 6/\mathbb{F}_{43^6}$ . A quadratic twist is then another curve  $BLS6 - 6_{2-twist}$  over  $\mathbb{F}_{43^3}$  isomorphic to the original curve. We use XXX. The task is to find an  $\omega \in \mathbb{F}_{43^6}$ , such that  $\omega^2 \in \mathbb{F}_{43^3}$ . We choose  $\omega = x^4 + 7x^3 + 9x^2 + 11x + 8$ . Then we interpret  $\delta = \omega^2 = 27x^2 + 17x + 35$  as an element from  $\mathbb{F}_{43^3}$ . So our twisted curve is  $y^2 = x^3 + a\delta^2x + b\delta^3 = x^3 + 6 \cdot (27t^2 + 17t + 35)$  so we get

$$BLS6 - 6_{2-twist}/\mathbb{F}_{43^3} : y^2 = x^3 + (10t^2 + 14t + 15)$$

## Baby JubJub

To give an understanding what the Baby-JubJub curve is, we want to parallel its development here to find a Baby-Jubjub like curve for pen and paper.

As with the original large Baby-JubJub curve we apply the method from to define a pen& paper Baby-JubJub-like curve over the scalar field of the "large" *BLS6* prime order subgroup, which is  $\mathbb{F}_{13}$ .

Since  $13 \bmod 4 = 1$  we would go with A.1. As we will only find a few curves, we will tweak the algorithm and run

```
sage: F13 = GF(13) 109
sage: for A in xrange(3, 13): 110
.....:     if (A-2) % 4 != 0: 111
.....:         continue 112
.....:     try: 113
.....:         E = EllipticCurve(F13, [0, A, 0, 1, 0]) # 114
Montgomery form
```

```

.....:      E                                     115
.....:      E.order()                             116
.....:      except:                               117
.....:      continue                              118

```

So we get two curves in Montgomery form  $y^2 = x^3 + 6 \cdot x^2 + x$  which has order 8 and  $y^2 = x^3 + 10 \cdot x^2 + x$ , which has order 16. We could transform one of them into an Edwards curve, however

So to find our Edwards curve, we will do exhaustive search rather

```

sage: for d in F13:                               119
.....:     j= ZZ(0)                               120
.....:     for x in F13:                           121
.....:         for y in F13:                       122
.....:             if x^2+y^2 == 1+d*x^2*y^2:      123
.....:                 j=j+1                        124
.....:         print('d=', d)                      125
.....:         print('order=', j)                  126

```

and get  $x^2 + y^2 = 1 + 7 \cdot x^2 y^2$  which has 20 points. The associated Montgomery curve is then using XXX given by  $8y^2 = x^3 + 6 \cdot x^2 + x$ .

So we define our Baby-JubJub Edwards curve to be

$$EdBJJ/\mathbb{F}_{13} : x^2 + y^2 = 1 + 7 \cdot x^2 y^2$$

with associated Montgomery form to be

$$MBJJ/\mathbb{F}_{13} : 8y^2 = x^3 + 6 \cdot x^2 + x$$

As  $20 = 2 \cdot 2 \cdot 5$ , we have a "large" prime order subgroup of order 5 and a cofactor 4. The group of rational points is

```

sage: for x in F13:                               127
.....:     for y in F13:                         128
.....:         if x^2+y^2 == F13(1)+F13(7)*x^2*y^2: 129
.....:             print(x,y)                     130

```

$(0, 1), (0, 12), (1, 0), (2, 4), (2, 9), (4, 2), (4, 11), (5, 6), (5, 7), (6, 5), (6, 8), (7, 5), (7, 8), (8, 6), (8, 7), (9, 2), (9, 11)$

with neutral element  $(0, 12)$

As expected we have a prime order subgroup of size 5, which can be generated by  $(11, 9)$ . We get  $\{(11, 9) \rightarrow (6, 8) \rightarrow (7, 8) \rightarrow (2, 9) \rightarrow (0, 1)\}$ .

```

sage: def Edwards_add((x1,y1),(x2,y2),d):         131
.....:     x3 = F13((F13(x1)*F13(y2)+F13(y1)*F13(x2))/((F13  132
.....:         (1)+F13(d)*F13(x1)*F13
.....:         (x2)*F13(y1)*F13(y2))))              133
.....:     y3 = F13((F13(y1)*F13(y2)-F13(x1)*F13(x2))/((F13  134
.....:         (1)-F13(d)*F13(x1)*F13
.....:         (x2)*F13(y1)*F13(y2))))              135
.....:     return (x3,y3)                          136

```

**Hashing to the pairing groups** We give various constructions to hash into  $\mathbb{G}_1$  and  $\mathbb{G}_2$ .

We start with hashing to the scalar field... TO APPEAR

Non of these techniques work for hashing into  $\mathbb{G}_2$ . We therefore implement Pederson's Hash for BLS6.

We start with  $\mathbb{G}_1$ . Our goal is to define an 12-bit bounded hash function

$$H_1 : \{0, 1\}^8 \rightarrow \mathbb{G}_1$$

Since  $12 = 3 \cdot 4$  we "randomly" select 4 uniformly distributed generators  $\{(38, 15), (35, 28), (27, 34), (38, 28)\}$  from  $\mathbb{G}_1$  and use the pseudo-random function from XXX. For every genrator we therefore have to choose a set of 4 randomly generated invertible elements from  $\mathbb{F}_{13}$ . We choose

$$\begin{aligned} (38, 15) & : \{2, 7, 5, 9\} \\ (35, 28) & : \{11, 4, 7, 7\} \\ (27, 34) & : \{5, 3, 7, 12\} \\ (38, 28) & : \{6, 5, 1, 8\} \end{aligned}$$

So our hash function is computed like this:

$$\begin{aligned} H_1(x_0, x_1, \dots, x_{11}) = & [2 \cdot 7^{x_0} \cdot 5^{x_1} \cdot 9^{x_2}](38, 15) + [11 \cdot 4^{x_3} \cdot 7^{x_4} \cdot 7^{x_5}](35, 28) + \\ & [5 \cdot 3^{x_6} \cdot 7^{x_7} \cdot 12^{x_8}](27, 34) + [6 \cdot 5^{x_9} \cdot 1^{x_{10}} \cdot 8^{x_{11}}](38, 28) \end{aligned}$$

Note that  $a^x = 1$  whe  $x = 0$  and hence those terms can be omitted in the computation. In particular the hash of the 12-bit zero string is given by

$$\begin{aligned} H_1(0) = & [2](38, 15) + [11](35, 28) + [5](27, 34) + [6](38, 28) = \\ & (27, 34) + (26, 34) + (35, 28) + (26, 9) = (33, 9) + (13, 28) = (38, 28) \end{aligned}$$

The hash of 011010101100 is given by

$$\begin{aligned} H_1(011010101100) = & [2 \cdot 7^0 \cdot 5^1 \cdot 9^1](38, 15) + [11 \cdot 4^0 \cdot 7^1 \cdot 7^0](35, 28) + [5 \cdot 3^1 \cdot 7^0 \cdot 12^1](27, 34) + [6 \cdot 5^1 \cdot 1^0 \cdot 8^0](38, 28) = \\ & [2 \cdot 5 \cdot 9](38, 15) + [11 \cdot 7](35, 28) + [5 \cdot 3 \cdot 12](27, 34) + [6 \cdot 5](38, 28) = \\ & [12](38, 15) + [12](35, 28) + [11](27, 34) + [4](38, 28) = \\ & \text{TO}_A\text{PPEAR} \end{aligned}$$

We can use the same technique to define a 12-bit bounded hash function in  $\mathbb{G}_2$ :

$$H_2 : \{0, 1\}^8 \rightarrow \mathbb{G}_2$$

Again we "randomly" select 4 uniformly distributed generators  $\{(7v^2, 16v^3), (42v^2, 16v^3), (17v^2, 15v^3), (10v^2, 15v^3)\}$  from  $\mathbb{G}_2$  and use the pseudo-random function from XXX. For every genrator we therefore have to choose a set of 4 randomly generated invertible elements from  $\mathbb{F}_{13}$ . We choose

$$\begin{aligned} (7v^2, 16v^3) & : \{8, 4, 5, 7\} \\ (42v^2, 16v^3) & : \{12, 1, 3, 8\} \\ (17v^2, 15v^3) & : \{2, 3, 9, 11\} \\ (10v^2, 15v^3) & : \{3, 6, 9, 10\} \end{aligned}$$

So our hash function is computed like this:

$$\begin{aligned} H_1(x_0, x_1, \dots, x_{11}) = & [8 \cdot 4^{x_0} \cdot 5^{x_1} \cdot 7^{x_2}](7v^2, 16v^3) + [12 \cdot 1^{x_3} \cdot 3^{x_4} \cdot 8^{x_5}](42v^2, 16v^3) + \\ & [2 \cdot 3^{x_6} \cdot 9^{x_7} \cdot 11^{x_8}](17v^2, 15v^3) + [3 \cdot 6^{x_9} \cdot 9^{x_{10}} \cdot 10^{x_{11}}](10v^2, 15v^3) \end{aligned}$$

## Baby-JubJub-2

To give an understanding what the Baby-JubJub curve is, we want to parallel its development here to find a Baby-JubJub like curve for pen and paper.

The original Baby-JubJub is a twisted Edwards curve over  $\mathbb{F}_7$  with  $a = -1$  and  $d = ?$ .

As with the original large Baby-JubJub curve we apply the method from to define a pen& paper Baby-JubJub-like curve over the scalar field of the "large" BLS6 prime order subgroup, which is  $\mathbb{F}_{13}$ .

Since  $13 \bmod 4 = 1$  we would go with A.1. As we will only find a few curves, we will tweak the algorithm and run

```
sage: F7 = GF(7) 137
sage: MNT4 = EllipticCurve (F7,[4 ,1]) 138
sage: [P.xy() for P in MNT4.points() if P.order() > 1] 139
[(0, 1), (0, 6), (4, 2), (4, 5)] 140
```

So we get two curves in Montgomery form  $y^2 = x^3 + 6 * x^2 + x$  which has order 8 and  $y^2 = x^3 + 10 * x^2 + x$ , which has order 16. We could transform one of them into an Edwards curve, however

So to find our Edwards curve, we will do exhaustive search rather

```
sage: F7t.<t> = F7[] 141
sage: F7_4.<u> = GF(7^4, name='u', modulus=t^4+t+1) # 142
    embedding degree is 4
sage: MNT4 = EllipticCurve (F7_4,[4 ,1]) 143
sage: INF = MNT4(0) # point at infinity 144
sage: for P in INF.division_points(5): # PI(P) == [q]P 145
....:     if P.order() == 5: # exclude point at infinity 146
....:         PiP = MNT4([a.frobenius() for a in P]) 147
....:         qP = 7*P 148
....:         if PiP == qP: 149
....:             print(P.xy()) 150
```

We want to choose a curve that has a large prime order subgroup and a small cofactor. So we go with  $2x^2 + y^2 = 1 + 3 \cdot x^2y^2$  which has order 14.

The associated Montgomery curve is then using XXX given by  $9y^2 = x^3 + 2x^2 + x$ .

So we define our Baby-JubJub Edwards curve to be

$$EdBJJ/\mathbb{F}_{13} : 2x^2 + y^2 = 1 + 3 \cdot x^2y^2$$

with associated Montgomery form to be

$$MBJJ/\mathbb{F}_{13} : 9y^2 = x^3 + 2x^2 + x$$

As  $14 = 2 \cdot 7$ , we have a "large" prime order subgroup of order 7 and a cofactor 2. The group of rational points is

```
sage: g1 = MNT4([0,1]) 151
sage: g2 = MNT4(2*u^3 + 5*u^2 + 4*u + 2, 2*u^3 + 3*u + 5) 152
sage: g1.weil_pairing(g2,5) 153
5*u^3 + 2*u^2 + 6*u 154
```

$(0, 1), (0, 12), (2, 4), (2, 9), (4, 5), (4, 8), (5, 2), (5, 11), (8, 2), (8, 11), (9, 5), (9, 8), (11, 4), (11, 9)$

with neutral element  $(0, 1)$

As expected we have a prime order subgroup of size 5, which can be generated by  $(11, 9)$ . We get  $\{(11, 9) \rightarrow (6, 8) \rightarrow (7, 8) \rightarrow (2, 9) \rightarrow (0, 1)\}$ .

```

sage: G.<x> = GF(5^6) # embedding degree is 6      155
sage: MNT6 = EllipticCurve (G,[2 ,1])             156
sage: INF = MNT6(0) # point at infinity            157
sage: for P in INF.division_points(7): # PI(P) == [q]P  158
.....:     if P.order() == 7: # exclude point at infinity  159
.....:         PiP = MNT6([a.frobenius() for a in P])      160
.....:         qP = 5*P                                     161
.....:         if PiP == qP:                                162
.....:             print(P.xy())                            163

```

## 4.8.2 MNT4 MNT6 Cycles

**Theorem 4.8.2.1.** *Let  $q$  be a prime and  $E/\mathbb{F}_q$  be an ordinary elliptic curve such that  $r = |E(Fq)|$  is a prime greater than 3.*

- *$E$  has embedding degree  $k = 4$  if and only if there exists  $x \in \mathbb{Z}$  such that  $t = -x$  or  $t = x + 1$ , and  $q = x^2 + x + 1$ .*
- *$E$  has embedding degree  $k = 6$  if and only if there exists  $x \in \mathbb{Z}$  such that  $t = 1 \pm 2x$  and  $q = 4x^2 + 1$ .*
- *There is an elliptic curve  $E/\mathbb{F}_q$  with embedding degree 6, discriminant  $D$ , and  $|E(Fq)| = r$  if and only if there is an elliptic curve  $E'/\mathbb{F}_r$  with embedding degree 4, discriminant  $D$ , and  $|E'(\mathbb{F}_r)| = q$ .*

We can use this theorem to find an MNT6-MNT4 cycle over very small prime fields with characteristics  $> 3$ :

**MNT4** For our MNT4 curve, we can choose  $x = 2$ . Then  $q = 7$  and if we choose  $t = x + 1$  then  $r = q + 1 - t = 7 + 1 - 3 = 5$ . Therefore our MNT4 curve is a curve  $y^2 = x^3 + ax + b$  defined over  $\mathbb{F}_7$  that consists of 5 points.

To construct the actual curve we could use the complex multiplication method again, but since the parameters  $a$  and  $b$  are from  $\mathbb{F}_7$  there are only 48 possibilities so we simply loop through all possible  $a$ 's and  $b$ 's and count the curve points until we find a curve that has 5 rational points. We get

$$y^2 = x^3 + 4x + 1$$

defined over  $\mathbb{F}_7$ , with scalar field  $\mathbb{F}_5$ . Since  $7 = 2^2 + 2 + 1$ , we know from theorem XXX, that this curve has embedding degree 4 and hence qualifies as a pen&paper pairing friendly elliptic curve. Since the curve's order is a prime and therefore has no non trivial factors, it has no non trivial subgroups. The curve has the following set of elements

$$MNT4 = \{(0, 1) \rightarrow (0, 6) \rightarrow (4, 2) \rightarrow (4, 5) \rightarrow \mathcal{O}\}$$



??The multiplication table is

$\cdot$	$\mathcal{O}$	$(0, 1)$	$(4, 5)$	$(4, 2)$	$(0, 6)$
$\mathcal{O}$	$\mathcal{O}$	$(0, 1)$	$(4, 5)$	$(4, 2)$	$(0, 6)$
$(0, 1)$	$(0, 1)$	$(4, 5)$	$(4, 2)$	$(0, 6)$	$\mathcal{O}$
$(4, 5)$	$(4, 5)$	$(4, 2)$	$(0, 6)$	$\mathcal{O}$	$(0, 1)$
$(4, 2)$	$(4, 2)$	$(0, 6)$	$\mathcal{O}$	$(0, 1)$	$(4, 5)$
$(0, 6)$	$(0, 6)$	$\mathcal{O}$	$(0, 1)$	$(4, 5)$	$(4, 2)$

In what follows we choose our generator to be  $g_{MNT4} = (0, 1)$ .

In what follows we want to compute type 2 pairings on our MNT4 curve. We therefore need to extract the subgroup  $\mathbb{G}_1$  as well as  $\mathbb{G}_2$  from the full 5-torsion group. Since the order of MNT4 is a prime number, we already know from XXX that  $\mathbb{G}_1$  is given by

$$\mathbb{G}_1 = \{(0, 1) \rightarrow (0, 6) \rightarrow (4, 2) \rightarrow (4, 5) \rightarrow \mathcal{O}\}$$

In type 2 pairings, the group  $\mathbb{G}_2$  is defined by those elements  $P$  of the full 5-torsion group, that are mapped to  $7 \cdot P$  under the Frobenius endomorphism XXX. Since  $MNT4/\mathbb{F}_{7^4}$  only contains 2475 elements, we can loop through all elements, to find the full 5-torsion group and extract all elements from  $\mathbb{G}_2$ : ??

Choose  $g_2 = (2u^3 + 5u^2 + 4u + 2, 2u^3 + 3u + 5)$  as generator of  $\mathbb{G}_2$ , we get

$$\mathbb{G}_2 = \{(2u^3 + 5u^2 + 4u + 2, 2u^3 + 3u + 5) \rightarrow (5u^3 + 2u^2 + 3u + 6, 2u^2 + 3u) \rightarrow (5u^3 + 2u^2 + 3u + 6, 5u^2 + 4u) \rightarrow (2u^3 + 5u^2 + 4u + 2, 5u^3 + 4u + 2) \rightarrow \mathcal{O}\}$$

e.g.  $[3]g_2 = (5u^3 + 2u^2 + 3u + 6, 5u^2 + 4u)$ .

Having those groups we can do pairings. We choose the Weil pairing and invoke `sagemath`. For example the Weil pairing between our two generators is

$$e(g_1, g_2) = 5u^3 + 2u^2 + 6u$$

??The full pairing table can the be written as

$e(\cdot, \cdot)$	$\mathcal{O}$	$g_1$	$[2]g_1$	$[3]g_1$	$[4]g_1$
$\mathcal{O}$	1	1	1	1	1
$g_2$	1	$5u^3 + 2u^2 + 6u$	$6u^3 + 5u^2 + 6$	$2u^3 + u^2 + 2u + 3$	$u^3 + 6u^2 + 6u + 4$
$[2]g_2$	1	$6u^3 + 5u^2 + 6$	$u^3 + 6u^2 + 6u + 4$	$5u^3 + 2u^2 + 6u$	$2u^3 + u^2 + 2u + 3$
$[3]g_2$	1	$2u^3 + u^2 + 2u + 3$	$5u^3 + 2u^2 + 6u$	$u^3 + 6u^2 + 6u + 4$	$6u^3 + 5u^2 + 6$
$[4]g_2$	1	$u^3 + 6u^2 + 6u + 4$	$2u^3 + u^2 + 2u + 3$	$6u^3 + 5u^2 + 6$	$5u^3 + 2u^2 + 6u$

**MNT6** For our MNT6 curve, we can choose  $x = 1$ . Then  $q = 5$  and if we choose  $t = 1 + 2x$  then  $r = q + 1 - t = 5 + 1 + 1 = 7$ . Therefore our MNT6 curve is a curve  $y^2 = x^3 + ax + b$  defined over  $\mathbb{F}_5$  that consists of 7 points.

To construct the actual curve we could use the complex multiplication method again, but since the parameters  $a$  and  $b$  are from  $\mathbb{F}_5$  there are only 24 possibilities, we simply

loop through all possible  $a$ 's and  $b$ 's and count the curve points until we find a curve that has 7 rational points. We get

$$y^2 = x^3 + 2x + 1$$

defined over  $\mathbb{F}_5$ . Since  $5 = 4 \cdot 1 + 1$ , we know from theorem XXX, that this curve has embedding degree 6 and hence qualifies as a pen&paper pairing friendly elliptic curve.

The curve has the following set of elements

$$MNT6 = \{(1, 2) \rightarrow (3, 3) \rightarrow (0, 1) \rightarrow (0, 4) \rightarrow (3, 2) \rightarrow (1, 3) \rightarrow \mathcal{O}\}$$

The multiplication table is

$\cdot$	$\mathcal{O}$	(1, 2)	(3, 3)	(0, 1)	(0, 4)	(3, 2)	(1, 3)
$\mathcal{O}$	$\mathcal{O}$	(1, 2)	(3, 3)	(0, 1)	(0, 4)	(3, 2)	(1, 3)
(1, 2)	(1, 2)	(3, 3)	(0, 1)	(0, 4)	(3, 2)	(1, 3)	$\mathcal{O}$
(3, 3)	(3, 3)	(0, 1)	(0, 4)	(3, 2)	(1, 3)	$\mathcal{O}$	(1, 2)
(0, 1)	(0, 1)	(0, 4)	(3, 2)	(1, 3)	$\mathcal{O}$	(1, 2)	(3, 3)
(0, 4)	(0, 4)	(3, 2)	(1, 3)	$\mathcal{O}$	(1, 2)	(3, 3)	(0, 1)
(3, 2)	(3, 2)	(1, 3)	$\mathcal{O}$	(1, 2)	(3, 3)	(0, 1)	(0, 4)
(1, 3)	(1, 3)	$\mathcal{O}$	(1, 2)	(3, 3)	(0, 1)	(0, 4)	(3, 2)

In what follows we choose our generator to be  $g_{MNT6} = (1, 2)$ .

In what follows we want to compute type 2 pairings on our MNT6 curve. We therefore need to extract the subgroup  $\mathbb{G}_1$  as well as  $\mathbb{G}_2$  from the full 7-torsion group. Since the order of MNT6 is a prime number, we already know from XXX that  $\mathbb{G}_1$  is given by

$$\mathbb{G}_1 = \{(1, 2) \rightarrow (3, 3) \rightarrow (0, 1) \rightarrow (0, 4) \rightarrow (3, 2) \rightarrow (1, 3) \rightarrow \mathcal{O}\}$$

In type 2 pairings, the group  $\mathbb{G}_2$  is defined by those elements  $P$  of the full 7-torsion group, that are mapped to  $5 \cdot P$  under the Frobenius endomorphism XXX. Since  $MNT6/\mathbb{F}_{5^6}$  contains 15680 elements, we can still loop through all elements, to find the full 7-torsion group and extract all elements from  $\mathbb{G}_2$

??

$$\begin{aligned} \mathbb{G}_2 = \{ & (x^3+2x^2+4x, x^5+2x^4+4x^3+3x^2+3) \rightarrow (x^5+4x^4+2x^3+3x^2+x+2, 3x^4+2x^3+x) \rightarrow \\ & (4x^5+x^4+2x^3, 3x^5+x^4+x^3+4x+4) \rightarrow (4x^5+x^4+2x^3, 2x^5+4x^4+4x^3+x+1) \rightarrow \\ & (x^5+4x^4+2x^3+3x^2+x+2, 2x^4+3x^3+4x) \rightarrow (x^3+2x^2+4x, 4x^5+3x^4+x^3+2x^2+2) \rightarrow \mathcal{O} \} \end{aligned}$$

We choose the generator  $g_2 = (x^3 + 2x^2 + 4x, x^5 + 2x^4 + 4x^3 + 3x^2 + 3)$

**Remark 18.** Note however that our MNT6 curve discriminant  $D = -16(4a^3 + 27b^2) = -16(4 \cdot 2^3 + 27 \cdot 1^2) = -944$ , while our MNT4 curve has discriminant XXX. Hence our example curves are not those guaranteed by theorem XXX. Those curve are both given by  $y^2 = x^3 + 2x + 1$  over  $\mathbb{F}_5$  and  $\mathbb{F}_7$ , respectively. However as both curves have the same defining equation, we rather choose examples that are visually distinguishable by their defining equations.

### 4.8.3 Edwards curve cycles

# Chapter 5

## Zk-Proof Systems

Some philosophical stuff about computational 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 it is correct. Is it possible to solely show the correctness of an assertion without revealing the knowledge of proofs? It turns out that it is indeed possible, and this is the topic of today's lecture: Zero Knowledge Systems.

**Example 29** (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  $\Sigma$  be a set and  $\Sigma^*$  the set of all finite strings of elements from  $\Sigma$ . Then a **formal language**  $L$  is a subset of  $\Sigma^*$ . The set  $\Sigma$  is called the **alphabet** of  $L$  and elements from  $L$  are called **words**. The rules that specify which strings from  $\Sigma^*$  belong to  $L$  are called the **grammar** of  $L$ .*

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

**Example 30** (Generalized factorization snark). *Consider example 29 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 29 is to simply find  $a, b, x, y \in \mathbb{F}_{13}$  such that  $x \cdot y \cdot a = b$ , 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 31** (Binary strings). *If we take the set  $\{0, 1\}$  as our alphabet  $\Sigma$  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 32** (Programming Language).

**Example 33** (Compiler).

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

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

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

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

**Remark 19.** *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 interpretation 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 34** (Generalized factorization snark). Consider example 29 and our associate formal language 30. 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 35** (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 36** (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

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

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  $\Sigma_I$  and  $\Sigma_W$  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  $f : (\Sigma_I \times \Sigma_W)^j \rightarrow \Sigma_I \times \Sigma_W$ .

**Remark 20** (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 satisfied if valid assignment (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 language...

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

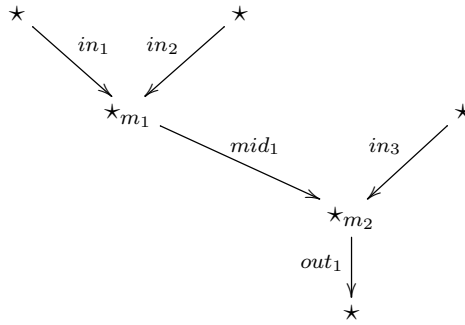
**Remark 21.** *Algebraic circuits are usually derived by Compilers, that transform higher languages to circuits. An example of such a compiler is XXX. Note: Different Compiler give very different circuit representations and Compiler optimization is important.*

**Example 37** (Generalized factorization snark). *Consider our generalized factorization example 29 with associated language 34.*

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

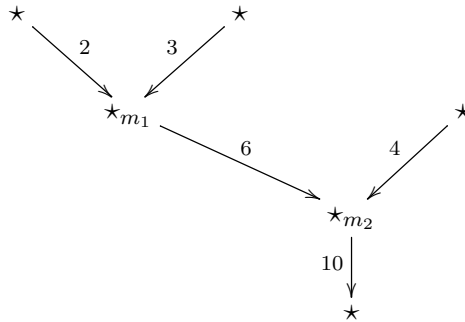
$$f : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \rightarrow \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto (x_1 \cdot x_2) \cdot x_3$$

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



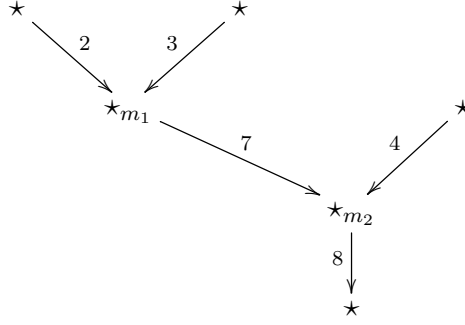
*with edge-index set  $I := \{in_1, in_2, in_3, mid_1, out_1\}$ .*

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



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

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



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

To match the requirements of the initial task 29, 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 than  $in_3$  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 38** (Baby JubJub for BLS6-6).

**Example 39** (ECDH as a circuit). over BLS6

**Example 40** (BLS Signature). example of one layer recursion over MNT4 and MNT6

**Example 41** (Boolean Circuits).

**Example 42** (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**  $\phi$  and **witness**  $w$ .

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

**Remark 22.** 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 43** (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  $6 \times 6$ -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 product we get two constraint equations

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

So from the way this R1CS is constructed, we know that whatever the underlying field  $\mathbb{F}$  is, the only solutions to this equations are

$$\{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$$

### 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  $m_1, \dots, m_k \in \mathbb{F}$  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  $t(x) := \prod_{l=1}^k (x - m_l)$  is a polynomial of 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

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



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 44** (Generalized factorization snark). *In this example we want to transform the R1CS from example 34 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*

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

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

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

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

- 
- $b = f(5) + 8m$
- $b = f(5) + 8 \cdot (7(f(7) + 12f(5)))$
- $b = f(5) + 4(f(7) + 12f(5))$
- $b = f(5) + 4f(7) + 9f(5)$
- $b = 10f(5) + 4f(7)$

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 polynomial divisibility we compute the polynomial  $p_I$  by

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

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 polynomial divisibility we compute the polynomial  $p_I$  by

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

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

## Boolean Algebra

Sometimes it is necessary to assume that a statement describes boolean variables. However by definition the alphabet of a statement is a finite field, which is often the scalar field of a large prime order cyclic group. So developers need a way to simulate boolean algebra inside other finite fields.

The most common way to do this, is to interpret the additive and multiply neutral element  $\{0, 1\} \subset F$  as boolean values. This is convinient because they are defined in any field.

**Boolean Constraint** So when a developer needs boolean variables as part of their statement, a R1CS is required on those variables, that enforces the variable to be either 1 or 0. So to "constrain a field element  $x \in \mathbb{F}$  to be 1 or 0 what we need is a system of equation  $(A_i x) \cdot (B_i x) = C_i x$  for some  $A_i, B_i, C_i \in \mathbb{F}$ , such that the only possible solutions for  $x$  are 0 or 1. As it turns out such a system can be realized by a single equation  $x \cdot (1 - x) = 0$  We see that indeed 0 and 1 are the only solutions here, since for the right side to be zero, at least one factor on the left side needs to be zero and this only happens for 0 and 1.

So now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix} \odot \begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix} = \begin{pmatrix} 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix}$$

So we get  $w = \begin{pmatrix} 1 \\ x \end{pmatrix}$  as well as  $A = \begin{pmatrix} 0 & 1 \end{pmatrix}$ ,  $B = \begin{pmatrix} 1 & -1 \end{pmatrix}$  and  $C = \begin{pmatrix} 0 & 0 \end{pmatrix}$ .

Once field elements are boolean constraint, we need constraints that are able to enforce boolean algebra on them. We therefore give constraints for the functionally complete set of Boolean operators give by *AND* and *NOT*. As all other boolean operations can be constructed from *AND* and *NOT*, this suffices. However in actual implementations it is of high importance to limit the number of constraints as much as possible. In reality it is therefor advantageous to implement all logic operators in constraints.

**AND-constraints** Given three field elements  $x, y, z \in \mathbb{F}$  that represent boolean variables, we want to find a R1CS, such that  $w = (1, x, y, z)$  satisfies the constraint system if and only if  $x \text{ AND } y = z$ .

So first we have to constrain  $x, y$  and  $z$  to be boolean as explained in XXX. The next thin is we need to find a R1CS that enforces the *AND* logic. We can simply choose  $x \cdot y = z$ , since (for boolean constraint values)  $x \cdot y$  equals 1 if and only if both  $x$  and  $y$  are 1.

Now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} \odot \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix}$$

Combining this R1CS with the required fthree boolean constraints for  $x, y$  and  $z$  we get

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} \odot \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix}$$

So from the way this R1CS is constructed, we know that whatever the underlying field  $\mathbb{F}$  is, the only solutions to this equations are

$$\{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$$

which is the set of all  $(x, y, z) \in \{0, 1\}^3$  such that  $x \text{ AND } y = z$ .

**NOT constraint** Given two field elements  $x, y \in \mathbb{F}$  that represent boolean variables, we want to find a R1CS, such that  $w = (1, x, y)$  satisfies the constraint system if and only if  $x = \neg y$ .

So again we have to constrain  $x$  and  $y$  to be boolean as explained in XXX. The next think is we need to find a R1CS that enforces the *NOT* logic. We can simply choose  $(1 - x) = y$ , since (for boolean constraint values) this enforces that  $y$  is always the boolean opposite of  $x$ .

Now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix} 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

So actually we wrote the linear equation  $1 - x = y$  like  $(1 - x) \cdot 1 = y$  and translated that into the matrix equation.

Combining this R1CS with the required fthree boolean constraints for  $x, y$  and  $z$  we get

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

So from the way this R1CS is constructed, we know that whatever the underlying field  $\mathbb{F}$  is, the only solutions to this equations are

$$\{(0, 1), (1, 0)\}$$

which is the set of all  $(x, y) \in \{0, 1\}^2$  such that  $x = \neg y$ .

**EXERCISE:** DO OR; XOR; NAND

More complicated logical constraints can then be obtained by combining all sub-R1CS together. For example if the task is to enforce  $(in_1 \text{ AND } \neg in_2) \text{ AND } in_3 = out_1$  we first apply the FLATTENING technique from XXX, which gives is

$$\begin{aligned} \neg in_2 &= mid_1 \\ in_1 \text{ AND } mid_1 &= mid_2 \\ mid_2 \text{ AND } in_3 &= out_1 \end{aligned}$$

So we have the statement  $w = (1, in_1, in_2, in_3, mid_1, mid_2, out_1)$ , 6 boolean constraints for the variables, 2 constraints for the 2 *AND* operations and 1 constraint for the *NOT* operation.

## Binary representations

In circuit computations its is often necessary to use the binary representation of a prime field element. Binary representations of prime field elements work exactly like binary representations of ordinary unsigned integers. Only the algebraic operations are different. To compute the binary representation of some number  $x \in \mathbb{F}_p$  we need to know the number of bits in the binary representation of  $p$  first. We write this as  $m = |p_{bin}|$ .

Then a bitstring  $(b_0, \dots, b_m) \in \{0, 1\}^m$  is the binary representation of the field element  $x$ , if and only if

$$x = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_m \cdot 2^m$$

Note that, since  $p$  is a prime number that has a leading bit 1 at position  $m$ . Moreover every prime number  $p > 2$  is odd and hence has least significant bit set to 1. Hence all numbers  $2^j$  for  $0 \leq j \leq m$  are elements of  $\mathbb{F}_p$  and the equation is well defined. We can therefore enforce this equation as a R1CS, by flattening the equation:

$$\begin{aligned} b_0 \cdot 1 &= mid_0 \\ b_1 \cdot 2 &= mid_1 \\ \dots &= \dots \\ b_m \cdot 2^m &= mid_m \\ (mid_0 + mid_1 + \dots + mid_m) \cdot 1 &= x \end{aligned}$$

So we have the statement  $w = (1, x, b_0, \dots, b_m, mid_0, \dots, mid_m)$  and we need  $(m + 1)$  constraints to enforce the binary representation in addition to the  $m$  constraints that enforce booleanness.

At this point we see, that writing more complex R1CS becomes clumsy and in actual implementations people therefore use languages to makes the constraint system more readable. In this example we could write for example something like this:

**Require:**  $m$  Bitlength of modulus

**Statement:**  $w \leftarrow [x, b[m], mid[m]]$

$tmp \leftarrow 0$

**for**  $j \leftarrow 1, \dots, m$  **do**

```

Constrain:  $b[j] \cdot (1 - b[j]) == 0$ 
Constrain:  $b[j] \cdot 2^j == mid[j]$ 
 $tmp = tmp + mid[j]$ 
end for
Constrain:  $tmp \cdot 1 == x$ 

```

keeping in mind that this is a meta level algorithm to **generate** the R1CS, not the R1CS itself, as constructs like for loops have not direct meaning on the level of the R1CS itself.

**Example 45.** *Considering the prime field  $\mathbb{F}_{13}$ , we want to enforce the binary representation of  $7 \in \mathbb{F}_{13}$ . To find the number of bits that we need to consider in our R1Cs, we start with the binary representation of 13, which is  $(1, 0, 1, 1)$  since  $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$ . So  $m = 4$  and we have to enforce a 4-bit representation for 7, which is  $(1, 1, 1, 0)$ , since  $7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$ .*

*A valid statement is then given by  $w = (1, 7, 1, 1, 1, 0, 1, 2, 4, 0)$  and indeed we satisfy the 9 required constraints*

$$\begin{array}{ll}
 1 \cdot (1 - 1) & = 0 \quad // \text{ boolean constraints} \\
 1 \cdot (1 - 1) & = 0 \\
 1 \cdot (1 - 1) & = 0 \\
 0 \cdot (1 - 0) & = 0 \\
 \\ 
 1 \cdot 1 & = 1 \\
 1 \cdot 2 & = 2 \\
 1 \cdot 4 & = 4 \\
 0 \cdot 8 & = 0 \\
 (1 + 2 + 4 + 0) \cdot 1 & = 7
 \end{array}$$

### Conditional (ternary) operator

It is often required to implement the ternary conditional operator  $? :$  as a R1CS. In general this operator takes three arguments, a boolean value  $b$  and two expressions  $if\_true$  and  $if\_false$ , usually written as  $b ? c : d$  and executes  $c$  and  $d$  according to the value of  $b$ .

If we assume all three arguments to be values from a finite field, such that  $b$  is boolean constraint (XXX), we can enforce a field element  $x$  to be the result of the conditional operator as

$$x = b \cdot c + (1 - b) \cdot d$$

Flattening the code gives

$$\begin{array}{ll}
 b \cdot c & = mid_0 \\
 (1 - b) \cdot d & = mid_1 \\
 (mid_0 + mid_1) \cdot 1 & = x
 \end{array}$$

So we have the statement  $w = (1, x, b, c, d, mid_0, mid_1)$  and we need 3 constraints to enforce the conditional operator in addition to 1 constraint that enforces booleanness of  $b$ .

NOTE: THERE WAS THIS PODCAST WITH ANNA AND THE GUY JAN TALKE TO WHERE HE SAID; CONDITIONALS CAN BE IMPLEMENTED SUCH THAT NOT BOTH BRANCHES ARE EXECUTED: LOOK THAT UP

## Range Proofs

$x > 5\dots$

## UintN

STUFF ABOUT HOW UINTN COMPUTATIONS ARE NOT STANDARDIZED AND THAT THERE ARE IMPLEMENTATIONS OTHER THEN MOD-N.... WE FIX ON MOD-N. WHAT DO ZEXE CIRCOM ECT FIX ON?

As we know circuits are not defined over integers but over finite fields instead. We therefore have no notation of integers in circuits. However on computers we also not use integers natively but Uint's instead.

As we know a UintN type is a representation of integers in the range of  $0 \dots 2^N$  with the exception that algebraic operations like addition and multiplication deviate from actual integers, whenever the result exceeds the largest representable number  $2^N - 1$ .

In circuit design it is therefore important to distinguish between various things tht might look like integers, but are actually not. For example Haskell's type NAT is an actual implementation of natural numbers. In particular this means ....

**Example 46** (Uint8). *What is  $0xFFF0 + 0xFFF0$  and so on...*

**Bit constraints** In prime fields, addition and multiplication behaves exactly like addition and multiplication with integers as long as the result does not exceed the modulus.

This makes the representation of UintNs in a prime field  $\mathbb{F}_p$  potentially ambiguous, as there are two possible representations, whenever  $2^N - 1 < p$ . In that case any element of *UintN* could be interpreted as an element of  $\mathbb{F}_p$ . This however is dangerous as the algebraic laws like addition and multiplication behave very different in general.

It is therefore common to represent UintN types in circuits as binary constraints strings of field elements of length  $N$ .

**Example 47.** *Consider the Uint4 type over the prime field  $\mathbb{F}_{17}$ . Since  $2^4 = 16$ , Uint4 can represent the numbers  $0, \dots, 15$  and it would be possible to interpret them as elements in  $\mathbb{F}_{17}$ . However addition*

## Twisted Edwards curves

Sometimes it required to do elliptic curve cryptography "inside of a circuit". This means that we have to implement the algebraic operations (addition, scalar multiplication) of an elliptic curve as a R1CS. To do this efficiently the curve that we want to implement must be defined over the same base field as the field that is used in the R1CS.

**Example 48.** *So for example when we consider an R1CS over the field  $\mathbb{F}_{13}$  as we did in example XXX, then we need a curve that is also defined over  $\mathbb{F}_{13}$ . Moreover it is advantageous to use a (twisted) Edwards curve inside a circuit, as the addition law contains no branching (See XXX). As we have seen in XXX our Baby-Jubjub curve is an Edwards curve defined over  $\mathbb{F}_{13}$ . So it is well suited for elliptic curve cryptography in our pend and paper examples*

**Twisted Edwards curves constraints** As we have seen in XXX, an Edwards curve over a finite field  $F$  is the set of all pairs of points  $(x, y) \in \mathbb{F} \times \mathbb{F}$ , such that  $x$  and  $y$  satisfy the equation  $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$ .

We can interpret this equation as a constraint on  $x$  and  $y$  and rewrite it as a R1CS by applying the flattenin technique from XXX.

$$\begin{aligned} x \cdot x &= x\_sq \\ y \cdot y &= y\_sq \\ x\_sq \cdot y\_sq &= xy\_sq \\ (a \cdot x\_sq + y\_sq) \cdot 1 &= 1 + d \cdot xy\_sq \end{aligned}$$

So we have the statement  $w = (1, x, y, x\_sq, y\_sq, xy\_sq)$  and we need 4 constraints to enforce that  $x$  and  $y$  are points on the Edwards curve  $x^2 + y^2 = 1 + d \cdot x^2 y^2$ . Writing the constraint system in matrix form, we get:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & a & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix} \odot \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & d \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix}$$

#### EXERCISE: WRITE THE R1CS FOR WEIERSTRASS CURVE POINTS

**Example 49** (Baby-JubJub). *Considering our pen and paper Baby JubJub curve over from XXX, we know that the curve is defined over  $\mathbb{F}_{13}$  and that  $(11, 9)$  is a curve point, while  $(2, 3)$  is not a curve point.*

*Starting with  $(11, 9)$ , we can compute the statement  $w = (1, 11, 9, 4, 3, 12)$ . Substituting this into the constraints we get*

$$\begin{aligned} 11 \cdot 11 &= 4 \\ 9 \cdot 9 &= 3 \\ 4 \cdot 3 &= 12 \\ (1 \cdot 4 + 3) \cdot 1 &= 1 + 7 \cdot 12 \end{aligned}$$

*which is true in  $\mathbb{F}_{13}$ . So our statement is indeed a valid assignment to the twisted Edwards curve constraining system.*

*Now considering the non valid point  $(2, 3)$ , we can still come up with some kind of statement  $w$  that will satisfy some of the constraints. But fixing  $x = 2$  and  $y = 3$ , we can never satisfy all constraints. For example  $w = (1, 2, 3, 4, 9, 10)$  will satisfy the first three constraints, but the last constrain can not be satisfied. Or  $w = (1, 2, 3, 4, 3, 12)$  will satisfy the first and the last constrain, but not the others.*

**Twisted Edwards curves addition** As we have seen in XXX one the major advantages of working with (twisted) Edwards curves is the existence of an addition law, that contains no branching and is valid for all curve points. Moreover the neutral element is not "at infinity" but the actual curve poin  $(0, 1)$ .

As we know from XXX, give two points  $(x_1, y_1)$  and  $(x_2, y_2)$  on a twisted Edwards curve their sum is given by

$$(x_3, y_3) = \left( \frac{x_1 y_2 + y_1 x_2}{1 + d \cdot x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a \cdot x_1 x_2}{1 - d \cdot x_1 x_2 y_1 y_2} \right)$$



We can realize this equation as a R1CS as follows: First not that we can rewrite the addition law as

$$\begin{aligned}
x_1 \cdot x_2 &= x_{12} \\
y_1 \cdot y_2 &= y_{12} \\
x_1 \cdot y_2 &= xy_{12} \\
y_1 \cdot x_2 &= yx_{12} \\
x_{12} \cdot y_{12} &= xy_{1212} \\
x_3 \cdot (1 + d \cdot xy_{1212}) &= xy_{12} + yx_{12} \\
y_3 \cdot (1 - d \cdot xy_{1212}) &= y_{12} - a \cdot x_{12}
\end{aligned}$$

So we have the statement  $w = (1, x_1, y_1, x_2, y_2, x_3, y_3, x_{12}, y_{12}, xy_{12}, yx_{12}, xy_{1212})$  and we need 7 constraints to enforce that  $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

**Example 50** (Baby-JubJub). *Considering our pen and paper Baby JubJub curve over from XXX. We recall from XXX that  $(11, 9)$  is a generator for the large prime order subgroup. We therefor already know from XXX that  $(11, 9) + (7, 8) = (11, 9) + [3](11, 9) = [4](11, 9) = (2, 9)$ . So we compute a valid statement as  $w = (1, 11, 9, 7, 8, 2, 9, 12, 7, 10, 11, 6)$ . Indeed*

$$\begin{aligned}
11 \cdot 7 &= 12 \\
9 \cdot 8 &= 7 \\
11 \cdot 8 &= 10 \\
9 \cdot 7 &= 11 \\
10 \cdot 11 &= 6 \\
2 \cdot (1 + 7 \cdot 6) &= 10 + 11 \\
9 \cdot (1 - 7 \cdot 6) &= 7 - 1 \cdot 12
\end{aligned}$$

There are optimizations for this using only 6 constraints, available:

**Twisted Edwards curves inversion** Similar to elliptic curves in Weierstrass form, inversion is cheap on Edwards curve as the negative of a curve point  $-(x, y)$  is given by  $(-x, y)$ . So a curve point  $(x_2, y_2)$  is the additive inverse of another curve point  $(x_1, y_1)$  precisely if the equation  $(x_1, y_1) = (-x_2, y_2)$  holds. We can write this as

$$\begin{aligned}
x_1 \cdot 1 &= -x_2 \\
y_1 \cdot 1 &= y_2
\end{aligned}$$

We therefor have a statement of the form  $w = (1, x_1, y_1, x_2, y_2)$  and can write the constraints into a matrix equation as

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix}$$

In addition we need the following constraints:

$$\begin{aligned}
x_1 \cdot 1 &= -x_2 \\
y_1 \cdot 1 &= y_2
\end{aligned}$$

**Twisted Edwards curves scalar multiplication** Although there are highly optimized R1CS implementations for scal multiplication on elliptic curves, the basic idea is somewhat simple: Given an elliptic curve  $E/\mathbb{F}_r$ , a scalar  $x \in \mathbb{F}_r$  with binary representation  $(b_0, \dots, b_m)$  and a curve point  $P \in E/\mathbb{F}_r$ , the scalar multiplication  $[x]P$  can be written as

$$[x]P = [b_0]P + [b_1]([2]P) + [b_2]([4]P) + \dots + [b_m]([2^m]P)$$

and since  $b_j$  is either 0 or 1,  $[b_j](kP)$  is either the neutral element of the curve or  $[2^j]P$ . However  $[2^j]P$  can be computed inductively by curve point doubling, since  $[2^j]P = [2]([2^{j-1}]P)$ .

So scalar multiplication can be reduced to a loop of length  $m$ , where the original curve point is repeatedly doubled and added to the result, whenever the appropriate bit in the scalar is equal to one.

So to enforce that a curve point  $(x_2, y_2)$  is the scalar product  $[k](x_1, y_1)$  of a scalar  $x \in \mathbb{F}_r$  and a curve point  $(x_1, y_1)$ , we need an R1CS that defines point doubling on the curve (XXX) and an R1CS that enforces the binary representation of  $x$  (XXX).

In case of twisted Edwards curve, we can use ordinary addition for doubling, as the constraints works for both cases (doubling is addition, where both arguments are equal). Moreover  $[b](x, y) = (b \cdot x, b \cdot y)$  for boolean  $b$ . Hence flattening equation XXX gives

$$\begin{aligned} b_0 \cdot x_1 &= x_{0,1} \quad // [b_0]P \\ b_0 \cdot y_1 &= y_{0,1} \end{aligned}$$

In addition we need to constrain  $(b_0, \dots, b_N)$  to be the binary representation of  $x$  and we need to constrain each  $b_j$  to be boolean.

As we can see a R1CS for scalar multiplication utilizes many R1CS that we have introduced before. For efficiency and readability it is therefore useful to apply the concept of a gadget (XXX). A pseudocode method to derive the associated R1CS could look like this:

```
Require:  $m$  Bitlength of modulus
Statement:  $w \leftarrow [x, b[m], mid[m]]$ 
   $tmp \leftarrow 0$ 
  for  $j \leftarrow 1, \dots, m$  do
    Constrain:  $b[j] \cdot (1 - b[j]) == 0$ 
    Constrain:  $b[j] \cdot 2^j == mid[j]$ 
     $tmp = tmp + mid[j]$ 
  end for
Constrain:  $tmp \cdot 1 == x$ 
```

**Curve Cycles** A particularly interesting case with far reaching implication is the situation when we have two curve  $E_1$  and  $E_2$ , such that the scalar field of curve  $E_1$  is the base field of curve  $E_2$  and vice versa. In that case it is possible to implement the group laws of one curve in circuits defined over the scalar field of the other curve.

## The RAM Model

FROM THE PODCAST WITH ANNA R. AND THE GUY FROM JAN...

## Generalizations

many circuits can be found here:

### 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  $\Sigma$ . 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

Preprocessing style: trusted setup, multi party ceremony

Blum, Feldman and Micali extended the notion of non-interactive zero-knowledge (NIZK) proofs in the common reference string model. NIZK proofs are useful in the construction 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  $\lambda$  in unary returns a polynomial time decidable binary relation  $R$ . For pairs  $(i, w) \in R$  we call  $i$  the instance<sup>1</sup> and  $w$  the witness. We define  $R_\lambda$  to be the set of possible relations  $R$  the relation generator may output given  $1^\lambda$ . We will in the following for notational simplicity assume  $\lambda$  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 (SETUP, PROVE, VFY, SIM) such*

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

#### 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 argument with a common reference string size proportional to the size of the statement and witness.

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 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.

Setup:

- random elements  $\alpha, \beta, \gamma, \delta, s \in \mathbb{F}_{\text{scalar}}$

---

<sup>1</sup>Note that in Groth16 this is called the statement. We think the term instance is more consistent with SOMETHING.

- Common reference string  $CRS_{QAP}$ , specific to the  $QAP$  and the choice of statement and witness  $CRS_{QAP} = (CRS_{\mathbb{G}_1}, CRS_{\mathbb{G}_2})$ , with  $n = \deg(t)$ :

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{l} [\alpha]g, [\beta]g, [\delta]g, \{[s^k]g\}_{k=0}^{n-1}, \left\{ \left[ \frac{\beta a_k(s) + \alpha b_k(s) + c_k(s)}{\gamma} \right] g \right\}_{k \in I} \\ \left\{ \left[ \frac{\beta a_k(s) + \alpha b_k(s) + c_k(s)}{\delta} \right] g \right\}_{k \in W}, \left\{ \left[ \frac{s^k t(s)}{\delta} \right] g \right\}_{k=0}^{n-2} \end{array} \right\}$$

$$CRS_{\mathbb{G}_2} = \left\{ [\beta]h, [\gamma]h, [\delta]h, \{[s^k]h\}_{k=0}^{n-1} \right\}$$

- Toxic waste: Must delete random elements after  $CRS_{QAP}$  generation.

**Example 51** (Generalized factorization snark). *In this example we want to compile our main example in Groth16. Input is the R1CS from example 43. We choose the following parameters*

$$\text{curve} = \text{BLS6-6} \quad \mathbb{G}_1 = \text{BLS6-6}(13) \quad g = (13, 15) \quad \mathbb{G}_2 = \quad h = (7v^2, 16v^3)$$

*Setup phase: Recall the quadratic arithmetic program of example XXX.*

*For our example we choose the following elements  $\alpha = 6$ ,  $\beta = 5$ ,  $\gamma = 4$ ,  $\delta = 3$ ,  $s = 2$  from  $\mathbb{F}_{13}$*

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{l} [6](13, 15), [5](13, 15), [3](13, 15), \{[s^k](13, 15)\}_{k=0}^1, \left\{ \left[ \frac{5a_k(2) + 6b_k(2) + c_k(2)}{4} \right] (13, 15) \right\}_{k \in S} \\ \left\{ \left[ \frac{5a_k(2) + 6b_k(2) + c_k(2)}{3} \right] (13, 15) \right\}_{k \in W}, \left\{ \left[ \frac{s^k t(2)}{3} \right] (13, 15) \right\}_{k=0}^0 \end{array} \right\}$$

*Since we have instance indices  $I = \{1, in_1, in_2\}$  and witness indices  $W = \{in_3, mid_1, out_1\}$  we have The instance parts.*

$$\left[ \frac{5a_c(2) + 6b_c(2) + c_c(2)}{4} \right] (13, 15) = \left[ \frac{5 \cdot 0 + 6 \cdot 0 + 0}{4} \right] (13, 15) = [0] (13, 15) = \mathcal{O}$$

$$\begin{aligned} \left[ \frac{5a_{in_3}(2) + 6b_{in_3}(2) + c_{in_3}(2)}{4} \right] (13, 15) &= [(5 \cdot 0 + 6 \cdot (7 \cdot 2 + 4) + 0) \cdot 10] (13, 15) = \\ &= [(6 \cdot 5) \cdot 10] (13, 15) = [1] (13, 15) = (13, 15) \end{aligned}$$

$$\begin{aligned} \left[ \frac{5a_{out}(2) + 6b_{out}(2) + c_{out}(2)}{4} \right] (13, 15) &= [(5 \cdot 0 + 6 \cdot 0 + (7 \cdot 2 + 4)) \cdot 10] (13, 15) = \\ &= [5 \cdot 10] (13, 15) = [11] (13, 15) = (33, 9) \end{aligned}$$

*Witness part:*

$$\begin{aligned} \left[ \frac{5a_{in_1}(2) + 6b_{in_1}(2) + c_{in_1}(2)}{3} \right] (13, 15) &= [(5 \cdot (6 \cdot 2 + 10) + 6 \cdot 0 + 0) \cdot 9] (13, 15) = \\ &= [(5 \cdot 9) \cdot 9] (13, 15) = [2] (13, 15) = (33, 34) \end{aligned}$$

$$\begin{aligned} \left[ \frac{5a_{in_2}(2) + 6b_{in_2}(2) + c_{in_2}(2)}{3} \right] (13, 15) &= [(5 \cdot 0 + 6 \cdot (6 \cdot 2 + 10) + 0) \cdot 9] (13, 15) = \\ &= [(6 \cdot 9) \cdot 9] (13, 15) = [5] (13, 15) = (26, 34) \end{aligned}$$

$$\left\lceil \frac{5a_{mid_1}(2) + 6b_{mid_1}(2) + c_{mid_1}(2)}{3} \right\rceil (13, 15) = [(5 \cdot (7 \cdot 2 + 4) + 6 \cdot 0 + 0) \cdot 9] (13, 15) =$$

$$[(5 \cdot 5) \cdot 9] (13, 15) = [4] (13, 15) = (35, 28)$$

For  $\left\{ \left\lceil \frac{s^k t(2)}{3} \right\rceil (13, 15) \right\}_{k=0}^0$  we get

$$\left\lceil \frac{2^0 t(2)}{3} \right\rceil (13, 15) = [t(2) \cdot 9] (13, 15) = [(2^2 + 2 + 9) \cdot 9] (13, 15) = [5] (13, 15) = (26, 34)$$

All together, the  $\mathbb{G}_1$  part of the CRS is:

$$CRS_{\mathbb{G}_1} = \left\{ (27, 34), (26, 34), (38, 15), \{(13, 15), (33, 34)\}, \{\mathcal{O}, (13, 15), (33, 9)\} \right. \\ \left. \{(33, 34), (26, 34), (35, 28)\}, \{(26, 34)\} \right\}$$

To compute the  $\mathbb{G}_2$  part

$$CRS_{\mathbb{G}_2} = \left\{ [5](7v^2, 16v^3), [4](7v^2, 16v^3), [3](7v^2, 16v^3), \{[2^k](7v^2, 16v^3)\}_{k=0}^1 \right\}$$

$$CRS_{\mathbb{G}_2} = \{ [5](7v^2, 16v^3), [4](7v^2, 16v^3), [3](7v^2, 16v^3), \{ [1](7v^2, 16v^3), [2](7v^2, 16v^3) \} \}$$

$$CRS_{\mathbb{G}_2} = \{ (16v^2, 28v^3), (37v^2, 27v^3), (42v^2, 16v^3), \{ (7v^2, 16v^3), (10v^2, 28v^3) \} \}$$

So altogether our common reference string is

$$\left( \left\{ (27, 34), (26, 34), (38, 15), \{(13, 15), (33, 34)\}, \{\mathcal{O}, (13, 15), (33, 9)\} \right. \right. \\ \left. \left. \{(33, 34), (26, 34), (35, 28)\}, \{(26, 34)\} \right\} \right. \\ \left. \{(16v^2, 28v^3), (37v^2, 27v^3), (42v^2, 16v^3), \{(7v^2, 16v^3), (10v^2, 28v^3)\} \} \right)$$

The proofer phase:

# Chapter 6

## Exercises and Solutions

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