

































Operational notes

















































Document updated on **March 19, 2022**.

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

















































- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- terms that have not yet been defined in the book
- text that needs advice from the communications/marketing team: Aaron & Shane
- text that needs to be completed or otherwise edited (by Sylvia)









































Todo list

10	 zero-knowledge proofs	12
11	 played with	12
12	 finite field	12
13	 elliptic curve	12
14	 Update reference when content is finalized	12
15	 methatical	12
16	 numerical	12
17	 a list of additional exercises	13
18	 think about them	13
19	 add some more informal explanation of absolute value	14
20	 We haven't really talked about what a ring is at this point	14
21	 What's the significance of this distinction?	15
22	 reverse	15
23	 Turing machine	15
24	 polynomial time	15
25	 sub-exponentially, with $\mathcal{O}((1 + \varepsilon)^n)$ and some $\varepsilon > 0$	15
26	 Add text	16
27	 \mathbb{Q} of fractions	16
28	 Division in the usual sense is not defined for integers	16
29	 Add more explanation of how this works	17
30	 pseudocode	18
31	 modular arithmetics	18
32	 actual division	18
33	 multiplicative inverses	18
34	 factional numbers	18
35	 exponentiation function	20
36	 See XXX	20
37	 once they accept that this is a new kind of calculations, its actually not that hard	20
38	 perform Euclidean division on them	20
39	 This Sage snippet should be described in more detail.	21
40	 prime fields	23
41	 residue class rings	23
42	 Algorithm sometimes floated to the next page, check this for final version	23
43	 Add a number and title to the tables	25
44	 (-1) should be (-a)?	26
45	 we have	28
46	 rephrase	32
47	 subtrahend	33
48	 minuend	33

49		what does this mean?	37
50		Chapter 1?	126
51		"rigorous"?	126
52		"proving"?	126
53		Add example	127
54		Add more explanation	127
55		I'd delete this, too distracting	127
56		binary tuples	127
57		add reference	128
58		add reference	128
59		check reference	128
60		check reference	128
61		Are we using w and x interchangeably or is there a difference between them?	129
62		check reference	129
63		jubjub	129
64		Edwards form	129
65		add reference	129
66		add reference	129
67		check wording	129
68		add reference	129
69		check references	130
70		add reference	130
71		add reference	130
72		preimage	131
73		check reference	131
74		add reference	131
75		check reference	132
76		check reference	132
77		add reference	133
78		Can we reword this? It's grammatically correct but hard to read	133
79		add reference	134
80		Schur/Hadamard product	134
81		add reference	134
82		check reference	134
83		check reference	135
84		add reference	136
85		check reference	137
86		check reference	137
87		check reference	137
88		check reference	137
89		check reference	138
90		add reference	138
91		add reference	139
92		check reference	139
93		check reference	139
94		add reference	140
95		add reference	140
96		add reference	141

97	■ We already said this in this chapter	143
98	■ check reference	143
99	■ add reference	143
100	■ check reference	144
101	■ add reference	144
102	■ check reference	144
103	■ Should we refer to R1CS satisfiability (p. 137 here?	145
104	■ add reference	146
105	■ add reference	146
106	■ add reference	146
107	■ add reference	147
108	■ check reference	147
109	■ check reference	148
110	■ check reference	150
111	■ add reference	151
112	■ "by"?	151
113	■ add reference	151
114	■ check reference	151
115	■ add reference	151
116	■ add reference	151
117	■ check reference	151
118	■ add reference	151
119	■ clarify language	153
120	■ add reference	154
121	■ add reference	154
122	■ add reference	154
123	■ add reference	154
124	■ add references	157
125	■ add references to these languages?	157
126	■ add reference	160
127	■ add reference	161
128	■ add reference	161
129	■ add reference	162
130	■ add reference	163
131	■ add reference	163
132	■ add reference	165
133	■ add reference	165
134	■ add reference	166
135	■ add reference	166
136	■ add reference	166
137	■ add reference	166
138	■ add reference	166
139	■ add reference	167
140	■ add reference	167
141	■ add reference	167
142	■ add reference	167
143	■ add reference	167
144	■ add reference	168

145	 add reference	169
146	 "constraints" or "constrained"?	169
147	 add reference	170
148	 "constraints" or "constrained"?	170
149	 add reference	170
150	 "constraints" or "constrained"?	170
151	 add reference	171
152	 add reference	171
153	 add reference	171
154	 add reference	171
155	 add reference	172
156	 add reference	173
157	 add reference	173
158	 add reference	173
159	 shift	175
160	 bishift	176
161	 add reference	177
162	 add reference	178
163	 something missing here?	179
164	 suggar	180
165	 add reference	180
166	 add reference	181
167	 add reference	182
168	 add reference	182
169	 add reference	182
170	 add reference	183
171	 add reference	183
172	 add reference	183
173	 add reference	184
174	 add reference	185
175	 add reference	186
176	 add reference	186
177	 add reference	186
178	 add reference	187
179	 add reference	187
180	 add reference	187
181	 add reference	187
182	 add reference	187
183	 "invariable"?	187
184	 add reference	188
185	 add reference	188
186	 add reference	188
187	 add reference	189
188	 add reference	189
189	 add reference	190
190	 add reference	191
191	 add reference	191
192	 add reference	192

193	 add reference	192
194	 add reference	192
195	 add reference	192
196	 add reference	192
197	 add reference	193
198	 add reference	193
199	 add reference	193
200	 add reference	193
201	 add reference	193
202	 add reference	193
203	 add reference	193
204	 add reference	193
205	 add reference	193
206	 add reference	194
207	 add reference	194
208	 add reference	194
209	 add reference	194
210	 add reference	196
211	 add reference	196
212	 add reference	196
213	 add reference	196
214	 add reference	196
215	 add reference	196
216	 add reference	197
217	 add reference	197
218	 add reference	197
219	 add reference	197
220	 add reference	197
221	 add reference	198
222	 add reference	198
223	 add reference	198
224	 add reference	198
225	 add reference	199
226	 add reference	199
227	 add reference	199
228	 add reference	199
229	 add reference	199
230	 add reference	199
231	 add reference	199
232	 add reference	199

233

MoonMath manual

234

TechnoBob and the Least Scruples crew

235

March 19, 2022

Contents

237	1 Introduction	5
238	1.1 Target audience	5
239	1.2 The Zoo of Zero-Knowledge Proofs	6
240	To Do List	8
241	Points to cover while writing	8
242	2 Preliminaries	9
243	2.1 Preface and Acknowledgements	9
244	2.2 Purpose of the book	9
245	2.3 How to read this book	10
246	2.4 Cryptological Systems	10
247	2.5 SNARKS	10
248	2.6 complexity theory	10
249	2.6.1 Runtime complexity	10
250	2.7 Software Used in This Book	11
251	2.7.1 Sagemath	11
252	3 Arithmetics	12
253	3.1 Introduction	12
254	3.1.1 Aims and target audience	12
255	3.1.2 The structure of this chapter	13
256	3.2 Integer Arithmetics	13
257	Euclidean Division	16
258	The Extended Euclidean Algorithm	18
259	3.3 Modular arithmetic	19
260	Congurency	20
261	Modular Arithmetics	20
262	The Chinese Remainder Theorem	23
263	Modular Inverses	26
264	3.4 Polynomial Arithmetics	29
265	Polynomial Arithmetics	33
266	Euklidean Division	34
267	Prime Factors	36
268	Lange interpolation	37
269	4 Algebra	40
270	4.1 Groups	40
271	Commutative Groups	41
272	Finite groups	43

273		Generators	43
274		The discrete Logarithm problem	43
275	4.1.1	Cryptographic Groups	44
276		The discret logarithm assumption	45
277		The decisional Diffi Hellman assumption	47
278		The computational Diffi Hellman assumption	47
279		Cofactor Clearing	48
280	4.1.2	Hashing to Groups	48
281		Hash functions	48
282		Hashing to cyclic groups	50
283		Hashing to modular arithmetics	51
284		Pederson Hashes	55
285		MimC Hashes	55
286		Pseudo Random Functions in DDH-A groups	55
287	4.2	Commutative Rings	55
288		Hashing to Commutative Rings	58
289	4.3	Fields	58
290		Prime fields	60
291		Square Roots	61
292		Exponentiation	63
293		Hashing into Prime fields	63
294		Extension Fields	63
295		Hashing into extension fields	66
296	4.4	Projective Planes	67
297	5	Elliptic Curves	69
298	5.1	Elliptic Curve Arithmetics	69
299	5.1.1	Short Weierstraß Curves	69
300		Affine short Weierstraß form	70
301		Affine compressed representation	74
302		Affine group law	75
303		Scalar multiplication	80
304		Projective short Weierstraß form	83
305		Projective Group law	85
306		Coordinate Transformations	85
307	5.1.2	Montgomery Curves	85
308		Affine Montgomery Form	87
309		Affine Montgomery coordinate transformation	88
310		Montgomery group law	90
311	5.1.3	Twisted Edwards Curves	90
312		Twisted Edwards Form	91
313		Twisted Edwards group law	92
314	5.2	Elliptic Curves Pairings	93
315		Embedding Degrees	93
316		Elliptic Curves over extension fields	95
317		Full Torsion groups	96
318		Torsion-Subgroups	98
319		The Weil Pairing	100

320	5.3	Hashing to Curves	103
321		Try and increment hash functions	103
322	5.4	Constructing elliptic curves	106
323		The Trace of Frobenius	106
324		The j -invariant	107
325		The Complex Multiplication Method	108
326		The <i>BLS6_6</i> pen& paper curve	117
327		Hashing to the pairing groups	124
328	6	Statements	126
329	6.1	Formal Languages	126
330		Decision Functions	127
331		Instance and Witness	130
332		Modularity	133
333	6.2	Statement Representations	133
334	6.2.1	Rank-1 Quadratic Constraint Systems	133
335		R1CS representation	134
336		R1CS Satisfiability	136
337		Modularity	138
338	6.2.2	Algebraic Circuits	138
339		Algebraic circuit representation	138
340		Circuit Execution	143
341		Circuit Satisfiability	145
342		Associated Constraint Systems	146
343	6.2.3	Quadratic Arithmetic Programs	151
344		QAP representation	151
345		QAP Satisfiability	153
346	7	Circuit Compilers	157
347	7.1	A Pen-and-Paper Language	157
348	7.1.1	The Grammar	157
349	7.1.2	The Execution Phases	159
350		The Setup Phase	159
351		The Prover Phase	161
352	7.2	Common Programing concepts	161
353	7.2.1	Primitive Types	161
354		The base-field type	162
355		The Subtraction Constraint System	165
356		The Inversion Constraint System	166
357		The Division Constraint System	167
358		The boolean Type	168
359		The boolean Constraint System	168
360		The AND operator constraint system	169
361		The OR operator constraint system	169
362		The NOT operator constraint system	170
363		Modularity	171
364		Arrays	174
365		The Unsigned Integer Type	174

366		The uN Constraint System	175
367		The Unsigned Integer Operators	176
368	7.2.2	Control Flow	177
369		The Conditional Assignment	177
370		Loops	179
371	7.2.3	Binary Field Representations	180
372	7.2.4	Cryptographic Primitives	182
373		Twisted Edwards curves	182
374		Twisted Edwards curves constraints	182
375		Twisted Edwards curve addition	183
376	8	Zero Knowledge Protocols	184
377	8.1	Proof Systems	184
378	8.2	The “Groth16” Protocol	185
379		The Setup Phase	187
380		The Proofer Phase	192
381		The Verification Phase	195
382		Proof Simulation	197
383	9	Exercises and Solutions	200

Chapter 1

Introduction

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

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

Zero-knowledge proofs (ZKPs) are an important privacy-enhancing tool from cryptography. 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.

1.1 Target audience

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

It is assumed that you have a basic understanding of programming and the underlying logical concepts. The first chapter covers the general syntax of Clarity but it does not delve into what

programming itself is all about. If this is what you are looking for, then you might have a more difficult time working through this book unless you have an (undiscovered) natural affinity for such topics. Do not let that dissuade you though, find an introductory programming book and press on! The straightforward design of Clarity makes it a great first language to pick up.

1.2 The Zoo of Zero-Knowledge Proofs

First, a list of zero-knowledge proof systems:

1. Pinocchio (2013): Paper

– Notes: trusted setup

2. BCGTV (2013): Paper

– Notes: trusted setup, implementation

3. BCTV (2013): Paper

– Notes: trusted setup, implementation

4. Groth16 (2016): Paper

– Notes: trusted setup

– Other resources: Talk in 2019 by Georgios Konstantopoulos

5. GM17 (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>
- Snark1: <http://ace.cs.ohio.edu/~gstewart/papers/snaark1.pdf>
- Virgo?: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F19/projects/reports/project5_report_ver2.pdf

Chapter 2

Preliminaries

2.1 Preface and Acknowledgements

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

2.2 Purpose of the book

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

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

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

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

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

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

2.3 How to read this book

Books and papers to read: XXXXXXXXXXXX

Software to try: XXXXXXXXXXXXXXXXXXXX

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

2.4 Cryptological Systems

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

2.5 SNARKS

2.6 complexity theory

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

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

2.6.1 Runtime complexity

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

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

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

For us, only a rough understanding of transit times is important in order to be able to talk about the security of 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.

STUFF ON CRYPTOGRAPHIC HASH FUNCTIOND

2.7 Software Used in This Book

2.7.1 Sagemath

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

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

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

¹<https://doc.sagemath.org/html/en/installation/index.html>

Chapter 3

Arithmetics

3.1 Introduction

3.1.1 Aims and target audience

The goal of this chapter is to enable a reader who is starting out with nothing more than basic high school algebra to be able to solve basic tasks in elliptic curve cryptography without the need of a computer.

How much mathematics do you need to understand **zero-knowledge proofs**? The answer, of course, depends on the level of understanding you aim for. It is possible to describe zero-knowledge proofs without using mathematics at all; however, to read a foundational paper like Groth [2016], some knowledge of mathematics is needed to be able to follow the discussion.

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

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

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

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

We want to stress, however, that this introduction is informal, incomplete and optimized to enable the reader to understand zero-knowledge concepts as efficiently as possible. Our focus and design choices are to include as little theory as necessary, focusing on the wealth of **numerical** examples. We believe that such an informal, example-driven approach to learning

zero-knowledge proofs

played with

finite field

elliptic curve

Update reference when content is finalized

methatical

numerical

mathematics may make it easier for beginners to digest the material in the initial stages.

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

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

a list of
additional
exercises

3.1.2 The structure of this chapter

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

After this practical warm up, we introduce some basic algebraic terms like groups and fields, because those terms are used very frequently in academic papers relating to zero-knowledge proofs. The beginner is advised to memorize those terms and **think about them**. We define these terms in the general abstract way of mathematics, hoping that the non mathematical trained reader will gradually learn to become comfortable with this style. We then give basic examples and do basic computations with these examples to get familiar with the concepts.

think
about
them

3.2 Integer Arithmetics

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

In what follows, we apply standard mathematical notations, and use the symbol \mathbb{Z} for the set of all **integers**: **S: I think it'd be useful to explain the difference between $:=$ and $=$ as well. We have a table on this in the ZKAPs whitepaper.**

M: Yeah maybe we use the more suggestive leftarrows aks \leftarrow ? If a table of symbols is unavoidable then ok, I find I super ugly, though

S: The table below is similar to what we have in the ZKAPs whitepaper. I think it's easier to read than a wall of text explaining the same things. We can make it more visually different (e.g. typesetting it in "dark mode", and probably move it earlier in the chapter.

Notation used in this chapter

Symbol	Meaning of Symbol	Example	Explanation
=	equals	$a = r$	a and r have the same value
:=	defining a variable	$\mathbb{Z} := \{\dots, -2, -1, 0, 1, 2, \dots\}$	\mathbb{Z} is the set of integers
\in	from the set	$a \in \mathbb{Z}$	a is an integer

$$\mathbb{Z} := \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\} \quad (3.1)$$

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

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

$$|4| = 4 \quad (3.2)$$

$$|-4| = 4 \quad (3.3)$$

add some more informal explanation of absolute value

In addition, we use the symbol \mathbb{N} for the set of all **counting numbers** (also called natural numbers). So whenever you see the symbol \mathbb{N} , think of the set of all non negative integers including the number 0:

$$\mathbb{N} := \{0, 1, 2, 3, \dots\} \quad (3.4)$$

Any number that is smaller than 0, that is, any number that has a minus sign, is not part of \mathbb{N} . All counting numbers are integers, but not the other way round. In other words, counting numbers are a subset of integers.

To make it easier to memorize new concepts and symbols, we might frequently link to definitions (See 3.1 for a definition of \mathbb{Z}) in the beginning, but as to many links render a text unreadable, we will assume the reader will become familiar with definitions as the text proceeds at which point we will not link them anymore.

Both sets \mathbb{N} and \mathbb{Z} have a notion of addition and multiplication defined on them. Most of us are probably able to do many integer computations in our head, but this gets more and more difficult as these increase in complexity. We will frequently invoke the SageMath system (2.7.1) for more complicated computations. One way to invoke the integer type in Sage is: **We haven't really talked about what a ring is at this point**

```

sage: ZZ # A sage notation for the integer type
Integer Ring
sage: NN # A sage notation for the counting number type
Non negative integer semiring
sage: ZZ(5) # Get an element from the Ring of integers
5
sage: ZZ(5) + ZZ(3)
8
sage: ZZ(5) * NN(3)
15
sage: ZZ.random_element(10**50)

```

add some more informal explanation of absolute value

We haven't really talked about what a ring is at this point

```

695 41259404776529716893829242545500390574195730349600 12
696 sage: ZZ(27713).str(2) # Binary string representation 13
697 110110001000001 14
698 sage: NN(27713).str(2) # Binary string representation 15
699 110110001000001 16
700 sage: ZZ(27713).str(16) # Hexadecimal string representation 17
701 6c41 18

```

One set of numbers that is of particular interest to us is **prime numbers**, which are counting numbers $p \in \mathbb{N}$ with $p \geq 2$, which are only divisible by themselves and by 1. All prime numbers apart from the number 2 are called **odd** (since even numbers greater than 2 are all divisible by 2, they are not prime numbers). We write \mathbb{P} for the set of all prime numbers and $\mathbb{P}_{\geq 3}$ for the set of all odd prime numbers. \mathbb{P} is infinite and can be ordered according to size, so that we can write them as follows:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, \dots \quad (3.5)$$

This is sequence A000040 in [OEIS](#), the On-Line Encyclopedia of Integer Sequences. In particular, we can talk about small and large prime numbers.

As the **fundamental theorem of arithmetics** tells us, prime numbers are, in a certain sense, the basic building blocks from which all other natural numbers are composed. To see that, let $n \in \mathbb{N}_{\geq 2}$ be any natural number. Then there are always prime numbers $p_1, p_2, \dots, p_k \in \mathbb{P}$, such that

$$n = p_1 \cdot p_2 \cdot \dots \cdot p_k. \quad (3.6)$$

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

Example 1 (Prime Factorization). To see what we mean by prime factorization of a number, let's look at the number $19214758032624000 \in \mathbb{N}$. To get its prime factors, we can successively divide it by all prime numbers in ascending order starting with 2:

$$19214758032624000 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 3 \cdot 5 \cdot 5 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 17 \cdot 23 \cdot 43 \cdot 43 \cdot 47$$

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

```

718 sage: n = NN(19214758032624000) 19
719 sage: factor(n) 20
720 2^7 * 3^3 * 5^3 * 7 * 11 * 17^2 * 23 * 43^2 * 47 21

```

This computation reveals an important observation: Computing the factorization of an integer is computationally expensive, while the reverse, that is, computing the product of given a set of prime numbers, is fast. MIRCO: inverse process? Its not reversing something actually, but my english doesn't resolve to this level very well

From this, an important question arises: How fast we can compute the prime factorization of a natural number? This is the famous **factorization problem** and, 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 algorithm known today run **sub-exponentially**, with $\mathcal{O}((1 + \varepsilon)^n)$ and some $\varepsilon > 0$.

It follows that number factorization \Leftrightarrow prime number multiplication is an example of a so-called **one-way function**: Something that is easy to compute in one direction, but hard to

What's the significance of this distinction?

reverse

Turing machine

polynomial time

sub-exponentially with $\mathcal{O}((1 + \varepsilon)^n)$ and

compute in the other direction. **The existence of one-way functions is a basic cryptographic assumptions that the security of many crypto systems is based on.**

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 that cryptosystems, which are based on the time complexity of the prime factor problem, are unsafe as soon as practically usable quantum computers become available. *Add text along the lines of "this is the best we got for now" Possibly something on when we can reasonably expect quantum computers to become accessible/usable enough*

Add text

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

Exercise 2. Compute the factorization of 6469693230 and double check your results using Sage.

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

1. The equation is defined over the type of natural numbers.

2. The equation is defined over the type of integers.

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

1. The equation is defined over the type of natural numbers.

2. The equation is defined over the type of integers.

3. The equation is defined over the type \mathbb{Q} of fractions.

 \mathbb{Q} of fractions

Euclidean Division *Division in the usual sense is not defined for integers*, as, for example, 7 divided by 3 will not be an integer again. However it is possible to divide any two integers with a remainder. So for example 7 divided by 3 is equal to 2 with a remainder of 1 , since $7 = 2 \cdot 3 + 1$.

Division in the usual sense is not defined for integers

Doing integer division like this is probably something many of us remember from school. It is usually called **Euclidean division**, or **division with a remainder**, and it is an essential technique to understand many concepts in this book. The precise definition is as follows:

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

$$a = m \cdot b + r \quad (3.7)$$

This decomposition of a given b is called **Euclidean division**, where a is called the **dividend**, b is called the **divisor**, m is called the **quotient** and r is called the **remainder**.

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

$$a \operatorname{div} b := m, \quad a \bmod b := r \quad (3.8)$$

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

So, in a nutshell Euclidean division is a process of dividing one integer by another in a way that produces a quotient and a non-negative remainder, the latter of which is smaller than the absolute value of the divisor. It can be shown, that both the quotient and the remainder always exist and are unique, as long as the dividend is different from 0.

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

Example 2. Applying Euclidean division and our previously defined notation 3.27 to the divisor -17 and the dividend 4 , we get

$$-17 \operatorname{div} 4 = -5, \quad -17 \operatorname{mod} 4 = 3$$

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

```

sage: ZZ(-17) // ZZ(4) # Integer quotient      22
-5                                             23
sage: ZZ(-17) % ZZ(4) # remainder             24
3                                             25
sage: ZZ(4).divides(ZZ(-17)) # self divides other 26
False                                         27
sage: ZZ(4).divides(ZZ(12))                  28
True                                          29

```

Methods to compute Euclidean division for integers are called **integer division algorithms**. Probably the best known algorithm is the so-called **long division**, which most of us might have learned in school. (It should be noted, however, that there are faster methods like **Newton–Raphson division**.)

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

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

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

Add more
explanation
of
how this
works

803 compute as follows

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

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

```

806 sage: ZZ(143785).quo_rem(ZZ(17)) # Euclidean Division      30
807 (8457, 16)                                                  31
808 sage: ZZ(143785) == ZZ(8457)*ZZ(17) + ZZ(16) # check      32
809 True                                                         33

```

810 *Exercise 5 (Integer Long Division).* Find an $m \in \mathbb{Z}$ as well as an $r \in \mathbb{N}$ such that $a = m \cdot b +$
 811 r holds for the following pairs $(a, b) = (27, 5)$, $(a, b) = (27, -5)$, $(a, b) = (127, 0)$, $(a, b) =$
 812 $(-1687, 11)$ and . In which cases are your solutions unique?

813 $(a, b) = (0, 7)$

814 *Exercise 6 (Long Division Algorithm).* Write an algorithm in pseudocode that computes integer pseudocode
 815 long division, handling all edge cases properly.

816 **The Extended Euclidean Algorithm** One of the most critical parts in this book is modular
 817 arithmetics XXX and its application in the computations in so-called **finite fields**, as we explain
 818 in XXX. In modular arithmetics, it is sometimes possible to define actual division and multi-
 819 plicative inverses of numbers, that is very different from inverses as we know them from other
 820 systems like factional numbers.

821 However, to actually compute those inverses, we have to get familiar with the so-called
 822 **extended Euclidean algorithm**. A few more terms are necessary to explain the concept: The
 823 **greatest common divisor** (GCD) of two nonzero integers a and b is the greatest non-zero
 824 counting number d such that d divides both a and b ; that is $d|a$ as well as $d|b$. We write
 825 $\gcd(a, b) := d$ for this number. In addition, two counting numbers are called **relative primes**
 826 or **coprimes**, if their greatest common divisor is 1.

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

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

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

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

modular
arith-
meticsactual
divisionmultiplicative
inversesfactional
numbers

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

836 *Example 4.* To illustrate the algorithm, let's apply it to the numbers $a = 12$ and $b = 5$. Since
 837 $12, 5 \in \mathbb{N}$ as well as $12 \geq 5$ all requirements are met and 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

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

842 **sage:** `ZZ(12).xgcd(ZZ(5)) # (gcd(a,b), s, t)` 34
 843 `(1, -2, 5)` 35

844 *Exercise 7* (Extended Euclidean Algorithm). Find integers $s, t \in \mathbb{Z}$ such that $\gcd(a, b) = s \cdot a +$
 845 $t \cdot b$ holds for the following pairs $(a, b) = (45, 10)$, $(a, b) = (13, 11)$, $(a, b) = (13, 12)$. What
 846 pairs (a, b) are coprime?

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

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

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

851 3.3 Modular arithmetic

852 In mathematics, **modular arithmetic** is a system of arithmetic for integers, where numbers
 853 "wrap around" when reaching a certain value, much like calculations on a clock wrap around
 854 whenever the value exceeds the number 12. For example, if the clock shows that it is 11 o'clock,

then 20 hours later it will be 7 o'clock, not 31 o'clock. The number 31 has no meaning on a normal clock that shows hours.

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

exponentiation
function

See XXX

Although modular arithmetic appears very different from ordinary integer arithmetic that we are all familiar with, we encourage the interested reader to work through the example and to discover that, **once they accept that this is a new kind of calculations, its actually not that hard.**

once they
accept
that this
is a new
kind of
calcula-
tions, its
actually
not that
hard

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

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

perform
Euclidean
division
on them

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

$$a \bmod n = b \bmod n \quad (3.11)$$

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

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

$$a \equiv b \pmod{n} \quad (3.12)$$

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

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

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

Suppose that the congruencies $a_1 \equiv b_1 \pmod{n}$ as well as $a_2 \equiv b_2 \pmod{n}$ are satisfied for integers $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and that $k \in \mathbb{Z}$ is another integer. Then:

$$\bullet a_1 + k \equiv b_1 + k \pmod{n} \text{ (compatibility with translation)}$$

$$\bullet k \cdot a_1 \equiv k \cdot b_1 \pmod{n} \text{ (compatibility with scaling)}$$

$$\bullet a_1 + a_2 \equiv b_1 + b_2 \pmod{n} \text{ (compatibility with addition)}$$

$$\bullet a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{n} \text{ (compatibility with multiplication)}$$

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

Note that the previous rules are implications, not equivalences, which means that you can not necessarily reverse those rules. The following rules makes this precise:

$$\bullet \text{ If } a_1 + k \equiv b_1 + k \pmod{n}, \text{ then } a_1 \equiv b_1 \pmod{n}$$

$$\bullet \text{ If } k \cdot a_1 \equiv k \cdot b_1 \pmod{n} \text{ and } k \text{ is coprime with } n, \text{ then } a_1 \equiv b_1 \pmod{n}$$

$$\bullet \text{ If } k \cdot a_1 \equiv k \cdot b_1 \pmod{k \cdot n}, \text{ then } a_1 \equiv b_1 \pmod{n}$$

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

$$k^p \equiv k \pmod{p}, \quad (3.13)$$

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

$$k^{p-1} \equiv 1 \pmod{p} \quad (3.14)$$

We can use Sage to compute examples for both k being coprime and not coprime to p :

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

```
1
```

```
sage: ZZ(64)**ZZ(137) % ZZ(137) == ZZ(64) % ZZ(137)
```

```
True
```

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

```
True
```

```
sage: ZZ(1918).gcd(ZZ(137))
```

```
137
```

```
sage: ZZ(1918)**ZZ(137) % ZZ(137) == ZZ(1918) % ZZ(137)
```

```
True
```

```
sage: ZZ(1918)**ZZ(137-1) % ZZ(137) == ZZ(1) % ZZ(137)
```

```
False
```

This Sage snippet should be described in more detail.

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

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

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

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

$$14x + 158 \equiv x - 102 \pmod{6}$$

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

$$\begin{aligned} 14x + 158 &\equiv x - 102 \pmod{6} \Leftrightarrow \\ 14x - x + 158 - 158 &\equiv x - x - 102 - 158 \pmod{6} \Leftrightarrow \\ 13x &\equiv -260 \pmod{6} \end{aligned}$$

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

$$13x \equiv -260 \pmod{6} \Leftrightarrow x \equiv -20 \pmod{6}$$

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

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

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

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

928 Putting all this together, we have shown that the every x from the set $\{x = 4 + k \cdot 6 \mid k \in \mathbb{Z}\}$ is a
 929 solution to the congruency $7 \cdot (2x + 21) + 11 \equiv x - 102 \pmod{6}$. We double ckeck for, say,
 930 $x = 4$ as well as $x = 14 + 12 \cdot 6 = 86$ using sage:

```
931 sage: (ZZ(7) * (ZZ(2) * ZZ(4) + ZZ(21)) + ZZ(11)) % ZZ(6) == (ZZ 48
932      (4) - ZZ(102)) % ZZ(6)
933 True 49
934 sage: (ZZ(7) * (ZZ(2) * ZZ(76) + ZZ(21)) + ZZ(11)) % ZZ(6) == ( 50
935      ZZ(76) - ZZ(102)) % ZZ(6)
936 True 51
```

937 Readers who had not been familiar with modular arithmetics until now and who might be
 938 discouraged by how complicated modular arithmetics seems at this point, should keep two

things in mind. First, computing congruencies in modular arithmetics is not really more complicated than computations in more familiar number systems (e.g. fractional numbers), it is just a matter of getting used to it. Second, the theory of prime fields (and more general residue class rings) takes a different view on modular arithmetics with the attempt to simplify matters. In other words, once we understand prime field arithmetics, things become conceptually cleaner and more easy to compute.

prime
fieldsresidue
class rings

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

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

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

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

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

has a solution, and all possible solutions of this congruence system are congruent modulo the product $N = n_1 \cdot \dots \cdot n_k$.¹ In fact, the following algorithm computes the solution set: **Algorithm** sometimes floated to the next page, check this for final version

Algorithm
sometimes
floated
to the
next page,
check this
for final
version

Algorithm 2 Chinese Remainder Theorem

Require: $n_0, \dots, n_{k-1} \in \mathbb{N}$ coprime

procedure CONGRUENCY-SYSTEMS-SOLVER($k, a_0, \dots, a_{k-1}, n_0, \dots, n_{k-1}$)

$N \leftarrow n_0 \cdot \dots \cdot n_{k-1}$

while $j < k$ **do**

$N_j \leftarrow N / n_j$

$(_, s_j, t_j) \leftarrow EXT - EUCLID(N_j, n_j)$

$$\triangleright 1 = s_j \cdot N_j + t_j \cdot n_j$$

end while

$x' \leftarrow \sum_{j=0}^{k-1} a_j \cdot s_j \cdot N_j$

$x \leftarrow x' \bmod N$

return $\{x + m \cdot N \mid m \in \mathbb{Z}\}$

end procedure

Ensure: $\{x + m \cdot N \mid m \in \mathbb{Z}\}$ is the complete solution set to 3.15.

¹This 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 but this is beyond the scope of this book. Interested readers should consult XXX [add references](#)

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

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

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

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

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. We can invoke Sage's computation of the Chinese Remainder Theorem (CRT) to double check our findings:

```
963 sage: CRT_list([4,1,3,0], [7,3,5,11]) 52
964 88 53
```

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

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

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

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

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

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

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 at the equivalence class of the result.

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

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

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

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

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

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

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

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

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

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

We can use the multiplication and addition table above to solve the equation on the right like we would solve normal integer equations: **Add a number and title to the tables**

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

Add a number and title to the tables

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

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

sage: `Z6 = Integers(6)`


```

1005 sage: Z6(2) + Z6(5) 55
1006 1 56
1007 sage: Z6(7) * (Z6(2) * Z6(4) + Z6(21)) + Z6(11) == Z6(4) - Z6(102) 57
1008 True 58

```

1009 *Jargon 1* (k -bit modulus). In cryptographic papers, we can sometimes read phrases like “[...] using a 4096-bit modulus”. This means that the underlying modulus n of the modular arithmetic used in the system has a binary representation with a length of 4096 bits. In contrast, the number 6 has the binary representation 110 and hence our example 8 describes a 3-bit modulus arithmetics system.

1014 *Exercise 16.* Let a, b, k be integers, such that $a \equiv b \pmod{n}$ holds. Show $a^k \equiv b^k \pmod{n}$.

1015 *Exercise 17.* Let a, n be integers, such that a and n are not coprime. For which $b \in \mathbb{Z}$ does the congruency $a \cdot x \equiv b \pmod{n}$ have a solution x and how does the solution set look in that case?

1018 **Modular Inverses** As we know, integers can be added, subtracted and multiplied so that the result is also an integer, but this is not true for the division of integers in general: for example, $3/2$ is not an integer anymore. To see why this is, from a more theoretical perspective, let us consider the definition of a multiplicative inverse first. When we have a set that has some kind of multiplication defined on it and we have a distinguished element of that set, that behaves neutrally with respect to that multiplication (doesn’t change anything when multiplied with any other element), then we can define **multiplicative inverses** in the following way:

1025 Let S be our set that has some notion $a \cdot b$ of multiplication and a **neutral element** $1 \in S$, such that $1 \cdot a = a$ for all elements $a \in S$. Then a **multiplicative inverse** a^{-1} of an element $a \in S$ is defined as follows:

$$a \cdot a^{-1} = 1 \quad (3.17)$$

1028 Informally speaking, the definition of a multiplicative inverse means that it “cancels” the original element to give 1 when they are multiplied.

1030 Numbers that have multiplicative inverses are of particular interest, because they immediately lead to the definition of division by those numbers. In fact, if a is number such that the multiplicative inverse a^{-1} exists, then we define **division** by a simply as multiplication by the inverse:

$$\frac{b}{a} := b \cdot a^{-1} \quad (3.18)$$

1034 *Example 9.* Consider the set of rational numbers, also known as fractions, \mathbb{Q} . For this set, the neutral element of multiplication is 1, since $1 \cdot a = a$ for all rational numbers. For example, $1 \cdot 4 = 4$, $1 \cdot \frac{1}{4} = \frac{1}{4}$, or $1 \cdot 0 = 0$ and so on.

1037 Every rational number $a \neq 0$ has a multiplicative inverse, given by $\frac{1}{a}$. For example, the multiplicative inverse of 3 is $\frac{1}{3}$, since $3 \cdot \frac{1}{3} = 1$, the multiplicative inverse of $\frac{5}{7}$ is $\frac{7}{5}$, since $\frac{5}{7} \cdot \frac{7}{5} = 1$, and so on.

1040 *Example 10.* Looking at the set \mathbb{Z} of integers, we see that with respect to multiplication the neutral element is the number 1 and we notice, that no integer $a \neq 1$ has a multiplicative inverse, since the equation $a \cdot x = 1$ has no integer solutions for $a \neq 1$.

1043 The definition of multiplicative inverse works verbatim for addition as well. In the case of integers, the neutral element with respect to addition is 0, since $a + 0 = a$ for all integers $a \in \mathbb{Z}$. The additive inverse always exist and is given by the negative number $-a$, since $a + (-a) = 0$.

(-1)
should
be (-a)?

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

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

Now, since 5 has a multiplicative inverse modulo 6, it makes sense to “divide” by 5 in \mathbb{Z}_6 . For example

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

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

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

$$1 = s \cdot n + t \cdot r \quad (3.19)$$

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

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

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

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

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

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

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

59
60

At this point, the attentive reader might notice that the situation where the modulus is a prime number is of particular interest, because we know from exercise XXX that in these cases all remainder classes must have modular inverses, since $\text{ggt}(r, n) = 1$ for prime n and $r < n$. In

fact, Fermat's little theorem provides a way to compute multiplicative inverses in this situation, since in case of a prime modulus p and $r < p$, we have

we have

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

1077 This tells us that the multiplicative inverse of a residue class r in modular p arithmetic is pre-
1078 cisely r^{p-2} .

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

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

1079 Addition and multiplication can be transferred to the equivalence classes, in a way exactly
1080 parallel to Example 8. This results in the following addition and multiplication tables:

	+	0	1	2	3	4		·	0	1	2	3	4
	0	0	1	2	3	4		0	0	0	0	0	0
1081	1	1	2	3	4	0		1	0	1	2	3	4
	2	2	3	4	0	1		2	0	2	4	1	3
	3	3	4	0	1	2		3	0	3	1	4	2
	4	4	0	1	2	3		4	0	4	3	2	1

1082 Calling the set of remainder classes in modular 5 arithmetics with this addition and multipli-
1083 cation \mathbb{F}_5 (for reasons we explain in more detail in XXX), we see some subtle but important
1084 differences to the situation in \mathbb{Z}_6 . In particular, we see that in the multiplication table, every
1085 remainder $r \neq 0$ has the entry 1 in its row and therefore has a multiplicative inverse. In addition,
1086 there are no non-zero elements such that their product is zero.

1087 To use Fermat's little theorem in \mathbb{F}_5 for computing multiplicative inverses (instead of using
1088 the multiplication table), let's consider $3 \in \mathbb{F}_3$. We know that the multiplicative inverse is given
1089 by the remainder class that contains $3^{5-2} = 3^3 = 3 \cdot 3 \cdot 3 = 4 \cdot 3 = 2$. And indeed $3^{-1} = 2$, since
1090 $3 \cdot 2 = 1$ in \mathbb{F}_5 .

1091 We can invoke Sage to do computations in our modular 5 arithmetics type to double-check
1092 our computations:

```

1093 sage: Z5 = Integers(5)                                     61
1094 sage: Z5(3) ** (5-2)                                         62
1095 2                                                            63
1096 sage: Z5(3) ** (-1)                                          64
1097 2                                                            65
1098 sage: Z5(3) ** (5-2) == Z5(3) ** (-1)                       66
1099 True                                                         67

```

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

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

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

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

1100 So, in this case, we cannot solve the equation for x by dividing by 3. And, indeed, when we look
 1101 at the multiplication table of \mathbb{Z}_6 (Example 8), we find that there are three solutions $x \in \{1, 3, 5\}$,
 1102 such that $3x + 3 = 0$ holds true for all of them.

1103 *Exercise 18.* Consider the modulus $n = 24$. Which of the integers 7, 1, 0, 805, -4255 have
 1104 multiplicative inverses in modular 24 arithmetics? Compute the inverses, in case they exist.

1105 *Exercise 19.* Find the set of all solutions to the congruency $17(2x + 5) - 4 \equiv 2x + 4 \pmod{5}$.
 1106 Then project the congruency into \mathbb{F}_5 and solve the resulting equation in \mathbb{F}_5 . Compare the results.

1107 *Exercise 20.* Find the set of all solutions to the congruency $17(2x + 5) - 4 \equiv 2x + 4 \pmod{6}$.
 1108 Then project the congruency into \mathbb{Z}_6 and try to solve the resulting equation in \mathbb{Z}_6 .

1109 3.4 Polynomial Arithmetics

1110 A polynomial is an expression consisting of variables (also called indeterminates) and coef-
 1111 ficients, that involves only the operations of addition, subtraction, multiplication, and non-
 1112 negative integer exponentiation of variables. All coefficients of a polynomial must have the
 1113 same type, e.g. being integers or fractions etc. To be more precise a *univariate polynomial* is
 1114 an expression

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

1115 where x is called the **indeterminate**, each a_j is called a **coefficient**. If R is the type of the
 1116 coefficients, then the set of all **univariate polynomials with coefficients in R** is written as
 1117 $R[x]$. We often simply use **polynomial** instead of univariate polynomial, write $P(x) \in R[x]$ for a
 1118 polynomial and denote the constant term as $P(0)$.

1119 A polynomial is called the **zero polynomial** if all coefficients are zero and a polynomial is
 1120 called the **one polynomial** if the constant term is 1 and all other coefficients are zero.

1121 If an univariate polynomial $P(x) = \sum_{j=0}^m a_j x^j$ is given, that is not the zero polynomial, we
 1122 call

$$\deg(P) := m \quad (3.21)$$

the *degree* of P and define the degree of the zero polynomial to be $-\infty$, where $-\infty$ (negative infinity) is a symbol with the property that $-\infty + m = -\infty$ for all counting numbers $m \in \mathbb{N}$. In addition, we write

$$Lc(P) := a_m \quad (3.22)$$

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

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

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

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

$$\begin{aligned} Q_1(x) &= 2x^2 + 4 + 3x^{-2} \\ Q_2(x) &= 0.5x^4 - 2x \\ Q_3(x) &= 1/x \end{aligned}$$

We can invoke Sage to do computations with polynomials. To do so, we have to specify the symbol for the indeterminate and the type for the coefficients. Note, however, that Sage defines the degree of the zero polynomial to be -1 .

```

1133 sage: Zx = ZZ['x'] # integer polynomials with indeterminate x 68
1134 sage: Zt.<t> = ZZ[] # integer polynomials with indeterminate t 69
1135 sage: Zx 70
1136 Univariate Polynomial Ring in x over Integer Ring 71
1137 sage: Zt 72
1138 Univariate Polynomial Ring in t over Integer Ring 73
1139 sage: p1 = Zx([17,-4,2]) 74
1140 sage: p1 75
1141 2*x^2 - 4*x + 17 76
1142 sage: p1.degree() 77
1143 2 78
1144 sage: p1.leading_coefficient() 79
1145 2 80
1146 sage: p2 = Zt(t^23) 81
1147 sage: p2 82

```

1148	t^23	83
1149	sage: p6 = Zx([0])	84
1150	sage: p6.degree()	85
1151	-1	86

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

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

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

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

1152 Again, we can use Sage to do computations with polynomials that have their coefficients in \mathbb{Z}_6 .
 1153 To do so, we have to specify the symbol for the indeterminate and the type for the coefficients:
 1154

1155	sage: Z6 = Integers(6)	87
1156	sage: Z6x = Z6['x']	88
1157	sage: Z6x	89
1158	Univariate Polynomial Ring in x over Ring of integers modulo 6	90
1159	sage: p1 = Z6x([5, -4, 2])	91
1160	sage: p1	92
1161	2*x^2 + 2*x + 5	93
1162	sage: p1 = Z6x([17, -4, 2])	94
1163	sage: p1	95
1164	2*x^2 + 2*x + 5	96
1165	sage: Z6x(x-2)*Z6x(x+3)*Z6x(x-5) == Z6x(x^3 + 2*x^2 + x)	97
1166	True	98

1167 Given some element from the same type as the coefficients of a polynomial, the polyno-
 1168 mial can be evaluated at that element, which means that we insert the given element for every
 1169 occurrence of the indeterminate x in the polynomial expression.

1170 To be more precise, let $P \in R[x]$, with $P(x) = \sum_{j=0}^m a_j x^j$ be a polynomial with a coefficient
 1171 of type R and let $b \in R$ be an element of that type. Then the **evaluation** of P at b is given by

$$P(a) = \sum_{j=0}^m a_j b^j \quad (3.23)$$

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

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

1172 Note, however, that is not possible to evaluate any of those polynomial on values of different
 1173 type. For example, it is strictly speaking wrong to write $P_1(0.5)$, since 0.5 is not an
 1174 integer. We can verify our computations using Sage:

1175	sage: <code>Zx = ZZ['x']</code>	99
1176	sage: <code>p1 = Zx([17, -4, 2])</code>	100
1177	sage: <code>p7 = Zx(x-2)*Zx(x+3)*Zx(x-5)</code>	101
1178	sage: <code>p1(ZZ(2))</code>	102
1179	<code>17</code>	103
1180	sage: <code>p7(ZZ(-6)) == ZZ(-264)</code>	104
1181	<code>True</code>	105

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

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

1182		
1183	sage: <code>Z6 = Integers(6)</code>	106
1184	sage: <code>Z6x = Z6['x']</code>	107
1185	sage: <code>p1 = Z6x([5, -4, 2])</code>	108
1186	sage: <code>p1(Z6(2)) == Z6(5)</code>	109
1187	<code>True</code>	110

Exercise 21. Compare both expansions of P_7 from $\mathbb{Z}[x]$ and from $\mathbb{Z}_6[x]$ in example XXX and example XXX, and consider the definition of \mathbb{Z}_6 as given in example XXX. Can you see how the definition of P_7 over \mathbb{Z} projects to the definition over \mathbb{Z}_6 if you consider the residue classes of \mathbb{Z}_6 ?

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

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

$$\sum_{n=0}^{m_1} a_n x^n + \sum_{n=0}^{m_2} b_n x^n = \sum_{n=0}^{\max(\{m_1, m_2\})} (a_n + b_n) x^n \quad (3.24)$$

$$\left(\sum_{n=0}^{m_1} a_n x^n \right) \cdot \left(\sum_{n=0}^{m_2} b_n x^n \right) = \sum_{n=0}^{m_1+m_2} \sum_{i=0}^n a_i b_{n-i} x^n \quad (3.25)$$

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

Regarding the definition of the degree of a polynomial, we see that the degree of the sum is always the maximum of the degrees of both summands, and the degree of the product is always the degree of the factors, since we defined $-\infty \cdot m = \infty$ for every integer $m \in \mathbb{Z}$. Using Sage's definition of degree, this would not hold, as the zero polynomials degree is -1 in Sage, which would violate this rule.

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

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

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

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

1208

1209	sage: <code>Zx = ZZ['x']</code>	111
1210	sage: <code>P = Zx([2, -4, 5])</code>	112
1211	sage: <code>Q = Zx([5, 0, -2, 1])</code>	113
1212	sage: <code>P+Q == Zx(x^3 + 3*x^2 - 4*x + 7)</code>	114

1213 **True** 115
 1214 **sage:** $P*Q == Zx(5*x^5 -14*x^4 +10*x^3+21*x^2-20*x +10)$ 116
 1215 **True** 117

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

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

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

1216
 1217 **sage:** $Z6x = \text{Integers}(6) ['x']$ 118
 1218 **sage:** $P = Z6x([2, -4, 5])$ 119
 1219 **sage:** $Q = Z6x([5, 0, -2, 1])$ 120
 1220 **sage:** $P+Q == Z6x(x^3 +3*x^2 +2*x +1)$ 121
 1221 **True** 122
 1222 **sage:** $P*Q == Z6x(5*x^5 +4*x^4 +4*x^3+3*x^2+4*x +4)$ 123
 1223 **True** 124

1224 *Exercise 22.* Compare the sum $P+Q$ and the product $P \cdot Q$ from the previous two examples
 1225 XXX and XXX and consider the definition of \mathbb{Z}_6 as given in example XXX. How can we derive
 1226 the computations in $\mathbb{Z}_6[x]$ from the computations in $\mathbb{Z}[x]$?

1227 **Euklidean Division** The ring of polynomials shares a lot of properties with integers. In par-
 1228 ticular, the concept of Euclidean division and the algorithm of long division is also defined for
 1229 polynomials. Recalling the Euclidean division of integers XXX, we know that, given two inte-
 1230 gers a and $b \neq 0$, there is always another integer m and a counting number r with $r < |b|$ such
 1231 that $a = m \cdot b + r$ holds.

1232 We can generalize this to polynomials whenever the leading coefficient of the dividend
 1233 polynomial has a notion of multiplicative inverse. In fact, given two polynomials A and $B \neq 0$
 1234 from $R[x]$ such that $Lc(B)^{-1}$ exists in R , there exist two polynomials M (the quotient) and R (the
 1235 remainder), such that

$$A = M \cdot B + R \quad (3.26)$$

1236 and $\deg(R) < \deg(B)$. Similarly to integer Euclidean division, both M and R are uniquely
 1237 defined by these relations.

1238 *Notation and Symbols 2.* Suppose that the polynomials A, B, M and R satisfy equation XX. Then
 1239 we often write

$$A \text{ div } B := M, \quad A \text{ mod } B := R \quad (3.27)$$

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

Require: $A, B \in R[x]$ with $B \neq 0$, such that $L_C(B)^{-1}$ exists in R

$$\begin{array}{l} M \leftarrow 0 \\ R \leftarrow A \\ d \leftarrow \deg(B) \\ c \leftarrow Lc(B) \\ \mathbf{while} \deg(R) \geq d \mathbf{do} \\ \quad S := Lc(R) \cdot c^{-1} \cdot x^{\deg(R)-d} \\ \quad M \leftarrow M + S \\ \quad R \leftarrow R - S \cdot B \end{array}$$
return (Q, R)

Ensure: $A = M \cdot B + R$

1248 *Example 21* (Polynomial Long Division). To give an example of how the previous algorithm
1249 works, let us divide the integer polynomial $A(x) = x^5 + 2x^3 - 9 \in \mathbb{Z}[x]$ by the integer polynomial
1250 $B(x) = x^2 + 4x - 1 \in \mathbb{Z}[x]$. Since B is not the zero polynomial and the leading coefficient of B
1251 is 1, which is invertible as an integer, we can apply algorithm 3. Our goal is to find solutions
1252 to equation XXX, that is, we need to find the quotient polynomial $M \in \mathbb{Z}[x]$ and the remainder
1253 polynomial $R \in \mathbb{Z}[x]$ such that $x^5 + 2x^3 - 9 = M(x) \cdot (x^2 + 4x - 1) + R$. Using a notation that is
1254 mostly used in Commonwealth countries, we compute as follows:

$$X^2 + 4X - 1) \begin{array}{r} X^3 - 4X^2 + 19X - 80 \\ X^5 + 2X^3 - 9 \\ -X^5 - 4X^4 + X^3 \\ \hline -4X^4 + 3X^3 \\ 4X^4 + 16X^3 - 4X^2 \\ \hline 19X^3 - 4X^2 \\ -19X^3 - 76X^2 + 19X \\ \hline -80X^2 + 19X - 9 \\ 80X^2 + 320X - 80 \\ \hline 339X - 89 \end{array} \quad (3.28)$$

35

```

1258 sage: Zx = ZZ['x']
1259 sage: A = Zx([-9, 0, 0, 2, 0, 1])
1260 sage: B = Zx([-1, 4, 1])
1261 sage: M = Zx([-80, 19, -4, 1])
1262 sage: R = Zx([-89, 339])
1263 sage: A == M*B + R
1264 True

```

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

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

Exercise 23. Consider the polynomial expressions $P(x) := -3x^4 + 4x^3 + 2x^2 + 4$ and $Q(x) = x^2 - 4x + 2$. Compute the Euclidean division of P by Q in the following types:

1. $P, Q \in \mathbb{Z}[x]$
2. $P, Q \in \mathbb{Z}_6[x]$
3. $P, Q \in \mathbb{F}_5[x]$

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

Exercise 24. Show that the polynomial $P(x) = 2x^4 - 3x + 4 \in \mathbb{F}_5[x]$ is a factor of the polynomial $Q(x) = x^7 + 4x^6 + 4x^5 + x^3 + 2x^2 + 2x + 3 \in \mathbb{F}_5[x]$, that is show $P|Q$. What is $Q \div P$?

Prime Factors Recall that the fundamental theorem of arithmetics XXX tells us that every number is the product of prime numbers. Something similar holds for polynomials, too.

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

$$P = F_1 \cdot F_2 \cdot \dots \cdot F_k. \quad (3.29)$$

This representation is unique, except for permutations in the factors and is called the **prime factorization** of P .

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

```

1296 sage: Zx = ZZ['x']
1297 sage: p = Zx(x^2-3)

```

```

1298 sage: p.roots()
1299 []
1300 sage: p.factor()
1301 x^2 - 3

```

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

Finding prime factors of a polynomial is hard. As we have seen in example XXX, points where a polynomial evaluates to zero, i.e points $x_0 \in R$ with $P(x_0) = 0$ are of special interest, since it can be shown the polynomial $F(x) = (x - x_0)$ is always a factor of P . The converse, however, is not necessarily true, because a polynomial can have irreducible prime factors.

Points where a polynomial evaluates to zero are called the **roots** of the polynomial. To be more precise, let $P \in R[x]$ be a polynomial. Then the set of all roots of P is defined as

$$R_0(P) := \{x_0 \in R \mid P(x_0) = 0\} \quad (3.30)$$

Finding the roots of a polynomial is sometimes called **solving the polynomial**. It is a hard problem and has been the subject of much research throughout history. In fact, it is well known that, for polynomials of degree 5 or higher, there is, in general, no closed expression, from which the roots can be deduced.

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

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

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

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

```

1322 sage: Zx = ZZ['x']
1323 sage: p = Zx(x^7 + 3*x^6 + 3*x^5 + x^4 - x^3 - 3*x^2 - 3*x - 1)
1324 )
1325 sage: p.roots()
1326 [(1, 1), (-1, 4)]
1327 sage: p.factor()
1328 (x - 1) * (x + 1)^4 * (x^2 + 1)

```

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

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

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

$$S = \{(x_0, y_0), (x_1, y_1), \dots, (x_m, y_m) \mid x_i \neq x_j \text{ for all indices } i \text{ and } j\} \quad (3.31)$$

This "few too many" property of polynomials is used in many places, like for example in erasure codes. It is also of importance in snarks and we therefore need to understand a method to actually compute a polynomial from a set of points.

what does this mean?

1335 If the coefficients of the polynomial we want to find have a notion of multiplicative inverse,
 1336 it is always possible to find such a polynomial. One method for this is called **Lagrange in-**
 1337 **terpolation**. It works as follows: Given a set like 3.31, a polynomial P of degree $m + 1$ with
 1338 $P(x_i) = y_i$ for all pairs (x_i, y_i) from S is given by the following algorithm:

Algorithm 4 Lagrange Interpolation

Require: R must have multiplicative inverses

Require: $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_m, y_m) \mid x_i, y_i \in R, x_i \neq x_j \text{ for all indices } i \text{ and } j\}$

procedure LAGRANGE-INTERPOLATION(S)

for $j \in (0 \dots m)$ **do**

$$l_j(x) \leftarrow \prod_{i=0; i \neq j}^m \frac{x - x_i}{x_j - x_i} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_m)}{(x_j - x_m)}$$

end for

$$P \leftarrow \sum_{j=0}^m y_j \cdot l_j$$

return P

end procedure

Ensure: $P \in R[x]$ with $\deg(P) = m$

Ensure: $P(x_j) = y_j$ for all pairs $(x_j, y_j) \in S$

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

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

1339 And, indeed, evaluation of P on the x -values of S gives the correct points, since $P(0) = 4$,
 1340 $P(-2) = 1$ and $P(2) = 3$.

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

from S . We can use the lookup tables XXX for computation in \mathbb{F}_5 and get the following:

$$l_0(x) = \frac{x-x_1}{x_0-x_1} \cdot \frac{x-x_2}{x_0-x_2} = \frac{x+2}{0+2} \cdot \frac{x-2}{0-2} = \frac{(x+2)(x-2)}{-4} = \frac{(x+2)(x+3)}{1} \\ = x^2 + 1$$

$$l_1(x) = \frac{x-x_0}{x_1-x_0} \cdot \frac{x-x_2}{x_1-x_2} = \frac{x-0}{-2-0} \cdot \frac{x-2}{-2-2} = \frac{x}{3} \cdot \frac{x+3}{1} = 2(x^2 + 3x) \\ = 2x^2 + x$$

$$l_2(x) = \frac{x-x_0}{x_2-x_0} \cdot \frac{x-x_1}{x_2-x_1} = \frac{x-0}{2-0} \cdot \frac{x+2}{2+2} = \frac{x(x+2)}{3} = 2(x^2 + 2x) \\ = 2x^2 + 4x$$

$$P(x) = 4 \cdot (x^2 + 1) + 1 \cdot (2x^2 + x) + 3 \cdot (2x^2 + 4x) \\ = 4x^2 + 4 + 2x^2 + x + x^2 + 2x \\ = 2x^2 + 3x + 4$$

1341 And, indeed, evaluation of P on the x -values of S gives the correct points, since $P(0) = 4$,
1342 $P(-2) = 1$ and $P(2) = 3$.

1343 *Exercise 25.* Consider example XXX and example XXX again. Why is it not possible to apply
1344 algorithm XXX if we consider S as a set of integers, nor as a set in \mathbb{Z}_6 ?

Chapter 4

Algebra

Todo: Def Subgroup, Fundamental theorem of cyclic groups.

We gave an introduction to the basic computational skills needed for a pen & paper approach to SNARKS in the previous chapter. In this chapter we get a bit more abstract and clarify a lot of mathematical terminology and jargon.

When you read papers about cryptography or mathematical papers in general, you will frequently stumble across algebraic terms like *groups*, *fields*, *rings* and similar. To understand what is going on, it is necessary to get at least some understanding of these terms. In this chapter we therefore with a short introduction to those terms.

In a nutshell, algebraic types like groups or fields define sets that are analog to numbers to various extend, in the sense that you can add, subtract, multiply or divide on those sets.

We know many example of sets that fall under those categories, like the natural numbers, the integers, the rational or the real numbers. they are in some sense already the most fundamental examples.

4.1 Groups

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

To understand groups, remember back in school when we learned about addition and subtraction of integers (Forgetting about integer multiplication for a moment). We learned that we can always add two integers and that the result is guaranteed to be an integer again. We also learned how to deal with brackets, 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 another integer (the negative), such that when we add both together we get zero.

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

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

A **group** (\mathbb{G}, \cdot) is a set \mathbb{G} , together with a map $\cdot : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}$, called the group law, such that the following properties hold:

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

- (Existence of an inverse) For every $g \in \mathbb{G}$ there is a $g^{-1} \in \mathbb{G}$, such that $g \cdot g^{-1} = e$ as well as $g^{-1} \cdot g = e$.
- (Associativity) For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.

Rephrasing the abstract definition in more layman's terms, a group is something, where we can do computations that resembles the behaviour of addition of integers. Therefore when the reader reads the term group they are advised to think of something where can combine some element with another element into a new element in a way that is reversible and where the order of combining many elements doesn't matter.

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

As we will see in what follows, groups are all over the place in cryptography and in SNARKS. In particular we will see in XXX, that the set of points on an elliptic curve define a group, which is the most important example in this book. To give some more familiar examples first:

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

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

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

To capture the special case of a group where the order in which the group law is executed doesn't matter, the concept of so called a **commutative group** is introduced. To be more precise a group is called commutative if $g_1 \cdot g_2 = g_2 \cdot g_1$ holds for all $g_1, g_2 \in \mathbb{G}$.

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

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

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

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

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

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

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

that takes pairs (g_1, g_2) (products) of elements from \mathbb{G}_1 and \mathbb{G}_2 and maps them somehow to elements from \mathbb{G}_3 , such that the *bilinearity* property holds: For all $g_1, g'_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ we have $e(g_1 + g'_1, g_2) = e(g_1, g_2) \cdot e(g'_1, g_2)$ and for all $g_1 \in \mathbb{G}_1$ and $g_2, g'_2 \in \mathbb{G}_2$ we have $e(g_1, g_2 + g'_2) = e(g_1, g_2) \cdot e(g_1, g'_2)$.

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

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

Example 32. Maybe the most basic example of a non-degenerate pairing is obtained, if we take \mathbb{G}_1 , \mathbb{G}_2 and \mathbb{G}_3 all to be the group of integers with addition $(\mathbb{Z}, +)$. Then the following map

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

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

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

Exercise 26. Consider example XXX again and let \mathbb{F}_5^* be the set of all remainder classes from \mathbb{F}_5 without the class 0. Then $\mathbb{F}_5^* = \{1, 2, 3, 4\}$. Show that (\mathbb{F}_5^*, \cdot) is a commutative group.

Exercise 27. Generalizing the previous exercise, consider general moduli n and let \mathbb{Z}_n^* be the set of all remainder classes from \mathbb{Z}_n without the class 0. Then $\mathbb{Z}_n^* = \{1, 2, \dots, n-1\}$. Give a counter example to show that (\mathbb{Z}_n^*, \cdot) is not a group in general.

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

Exercise 28. Consider the remainder class groups $(\mathbb{Z}_n, +)$ for some modulus n . Show that the map

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

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

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

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

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

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

Exercise 29. The RSA crypto system is based on a modulus n that is typically the product of two prime numbers of size 2048-bits. What is (approximately) the order of the remainder class group $(\mathbb{Z}_n, +)$ in this case?

Generators Of special interest, when working with groups are sets of elements that can generate the entire group, by applying the group law repeatedly to those elements or their inverses only.

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

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

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

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

The discrete Logarithm problem In cryptography in general and in snark development in particular, we often do computations "in the exponent" of a generator. To see what this means, observe, that when \mathbb{G} is a cyclic group of order n and $g \in \mathbb{G}$ is a generator of \mathbb{G} , then there is a map, called the **exponential map** with respect to the generator g

$$g^{(\cdot)} : \mathbb{Z}_n \rightarrow \mathbb{G} \quad x \mapsto g^x \quad (4.2)$$

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

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

Since the exponential map respects the group law, it doesn't matter if we do our computation in \mathbb{Z}_n before we write the result into the exponent of g or afterwards. The result will be the same. This is what is usually meant by saying we do our computations "in the exponent".

Example 37. Consider the multiplicative group (\mathbb{F}_5^*, \cdot) from example XXX. We know that \mathbb{F}_5^* is a cyclic group of order 4 and that every element is a generator. Choose $3 \in \mathbb{F}_5^*$, we then know that the map

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

respects the group law of addition in \mathbb{Z}_4 and the group law of multiplication in \mathbb{F}_5^* . And indeed doing a computation like

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

in the exponent gives the same result as doing the same computation in \mathbb{F}_5^* , that is

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

Since the exponential map is a one-to-one correspondence, that respects the group law, it can be shown that this map has an inverse

$$\log_g(\cdot) : \mathbb{G} \rightarrow \mathbb{Z}_n x \mapsto \log_g(x) \quad (4.3)$$

which is called the **discrete logarithm** map with respect to the base g . Discrete logarithms are highly important in cryptography as there are groups, such that the exponential map and its inverse the discrete logarithm, are believed to be one way functions, that is while it is possible to compute the exponential map in polynomial time, computing the discrete log takes (sub)-exponential time. We will look at this and similar problems in more detail in the next section.

4.1.1 Cryptographic Groups

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

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

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

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

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

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

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

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

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

$$h = g^x \quad (4.4)$$

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

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

Example 39 (Public key cryptography). One the most basic examples of an application for DL-A groups is in public key cryptography, where some pair (\mathbb{G}, g) is publically agreed on, such that \mathbb{G} is a finite cyclic group sufficiently large order r , where it is believed that the discrete logarithm assumption holds and g is a generator of \mathbb{G} .

In this setting a secret key is nothing but some number $sk \in \mathbb{Z}_r$ and the associated public key pk is the group element $pk = g^{sk}$. Since discrete logarithms are assumed to be hard it is therefore infisble for an attacker to compute the secret key from the public key, since it is believed to be hard to find solutions x to the equation

$$pk = g^x$$

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

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

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

We show that in this case \mathbb{Z}_p^* is not a DL-A group, by constructing an algorithm, which is able compute some $x \in \mathbb{Z}_{2^n}$ for any given generator g and arbitrary element h of \mathbb{F}_p^* , such that

$$h = g^x$$

holds and the run time complexity of the constructed algorithm is $\mathcal{O}(n^2)$, which is quadratic in the security parameter $n = \log_2(2^n)$.

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

A first thing to note is that since x is a number in modular 2^n arithmetic, we can write the binary representation of x as

$$x = c_0 \cdot 2^0 + c_1 \cdot 2^1 + \dots + c_n \cdot 2^n$$

with binary coefficients $c_j \in \{0, 1\}$. In particular x as an n -bit number, if interpreted as an integer.

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

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

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

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

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

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

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

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

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

It is easy to see that DDH-A is a stronger assumption then DL-A, in the sense that the discrete logarithm assumption is necessary for the decisional Diffi Hellman assumption to hold, but not the other way around.

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

On the other hand, the following example shows, that there are groups where the discrete logarithm assumption holds but the decisional Diffi Hellman assumption does not hold:

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

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

$$e(g^a, g^b) = e(g, z)$$

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

It follows that DDH-A is indeed weaker then DL-A and groups with efficient pairings can not be DDH-A groups. As the following example shows, another important class of groups, where DDH-A does not hold are the multiplicative groups of prime number residue classes.

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

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

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

In general it is not know if CDH-A is a stronger assumption then DL-A, or if both assumptions are equivalent. It is known that DL-A is necessary for CDH-A but the other direction is currently not well understood. In particular there are no groups known where DL-A holds but CDH-A does not hold.

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

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

From the CDH-A various variations and specializations are known. For example the so called *square computational Diffi-Hellman assumption* assumes, that given g and g^x it is computationally hard to compute g^{x^2} while the so called *inverse computational Diffi-Hellman assumption* assumes, that given g and g^x it is computationally hard to compute $g^{x^{-1}}$.

Cofactor Clearing TODO: (theorem: every factor of order defines a subgroup...)

4.1.2 Hashing to Groups

Hash functions Generally speaking, a hash function is any function that can be used to map data of arbitrary size to fixed-size values. Since binary strings of arbitrary length are a general way to represent arbitrary data, we can understand a general **hash function** as a map

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^k \quad (4.5)$$

where $\{0, 1\}^*$ represents the set of all binary strings of arbitrary but finite length and $\{0, 1\}^k$ represents the set of all binary strings that have a length of exactly k bits. So in our definition a hash function maps binary strings of arbitrary size onto binary strings of size exactly k . We call the images of H , that is the values returned by the hash function *hash values*, *digests*, or simply *hashes*.

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

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

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

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

In addition it should be infeasible to find two strings $x_1, x_2 \in \{0, 1\}^*$, such that $H(x_1) = H(x_2)$, which is called *collision resistance*. It is important to note though, that collisions always exist, since a function $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$ inevitably maps infinite many values onto the same hash. In fact, for any hash function with digests of length k , finding a preimage to a given digest can

always be done using a brute force search in 2^k evaluation steps. It should just be practically impossible to compute those values and statistically very unlikely to generate two of them by chance.

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

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

Example 44 (k -truncation hash). Considering the k -truncation hash from example XXX. Since the empty string has length 0 it follows that the digest of the empty string is string of length k that only contains 0's. i.e

$$H_k('') = (000 \dots 000)$$

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

In addition it is easy to construct collusions as all strings of size $> k$ that share the same k -bits "on the right" are mapped to the same hash value.

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

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

Example 45. An example of a hash function that is generally believed to be a cryptographically secure hash function is the so called *SHA256* hash, which in our notation is a function

$$SHA256 : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$$

that maps binary strings of arbitrary length onto binary strings of length 256. To evaluate a proper implementation of the *SHA256* hash function the digest of the empty string is supposed to be

$$SHA256('') = e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855$$

For better human readability it is common practise to represent the digest of a string, not in its binary form but in a hexadecimal representation. We can use sage to compute *SHA256* and freely transit between binary, hexadecimal and decimal representations. To do so we have to import for example hashlib's implementation of sha256:

```
sage: import hashlib 144
sage: test = 'e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934 145
       ca495991b7852b855'
sage: hasher = hashlib.sha256(b'') 146
sage: str = hasher.hexdigest() 147
```



```

1699 sage: type(str) 148
1700 <class 'str'> 149
1701 sage: d = ZZ('0x'+ str) # conversion to integer type 150
1702 sage: d.str(16) == str 151
1703 True 152
1704 sage: d.str(16) == test 153
1705 True 154
1706 sage: d.str(16) 155
1707 e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b8 156
1708 55
1709 sage: d.str(2) 157
1710 11100011101100001100010001000010100110001111110000011100000101 158
1711 0010011010111110111110100110010001001100101101111011110010
1712 01001000010011110101110010000011110010001100100100110111001
1713 00110100110010100100100101011001100100011011011110000101001
1714 01011100001010101
1715 sage: d.str(10) 159
1716 10298733624955409702953521232258132278979990064819803499337939 160
1717 7001115665086549

```

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

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

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

$$H : \{0, 1\}^* \rightarrow \mathbb{G} \quad (4.6)$$

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

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

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

To be more precise, choose an ordinary hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$ for some parameter k and a generator g of \mathbb{G} . Then the expression

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

is a positive integer, where $H(s)_j$ means the bit at the j -th position of $H(s)$. A hash-to-group function for the group \mathbb{G} can then be defined as a concatenation of the exponential map $g^{(\cdot)}$ of

g with the interpretation of $H(s)$ as an integer:

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

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

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

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

Applications where discrete log relations between hash values are undesirable therefore need different approaches and many of those approaches start with a way to hash into the sets \mathbb{Z}_r of modular r arithmetics.

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

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

$$z = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_{k-2} \cdot 2^{k-2} \quad (4.7)$$

Now since $z < r$, we know that z is guaranteed to be in the set $\{0, 1, \dots, r-1\}$ and hence can be interpreted as an element of \mathbb{Z}_r . From this follows that if $H : \{0, 1\}^* \rightarrow \{0, 1\}^{k-1}$ is a hash function, then a hash-to-group function can be constructed by

$$H_{r.\text{nbits}()-1} : \{0, 1\}^* \rightarrow \mathbb{Z}_r : s \mapsto H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \dots + H(s)_{k-2} \cdot 2^{k-2} \quad (4.8)$$

where $H(s)_j$ means the j 's bit of the image binary string $H(s)$ of the original binary hash function.

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

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

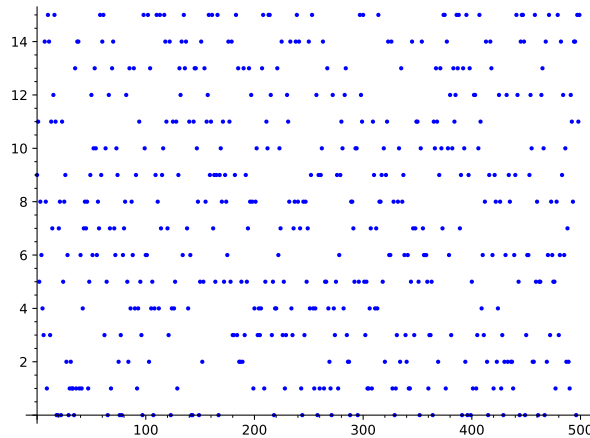
Example 47. To give an implementation of the $H_{r.nb\text{its}()-1}$ hash function, we use a 5-bit truncation of the *SHA256* hash from example XXX and define a hash into \mathbb{Z}_{16} by

$$H_{16.nb\text{its}()-1} : \{0, 1\}^* \rightarrow \mathbb{Z}_{16} : s \mapsto \text{SHA256}(s)_0 \cdot 2^0 + \text{SHA256}(s)_1 \cdot 2^1 + \dots + \text{SHA256}(s)_4 \cdot 2^4$$

1763 Since $k = 16.nb\text{its}() = 5$ and $16 - 2^{k-1} = 0$ this hash maps uniformly onto \mathbb{Z}_{16} . We can invoke
1764 sage to implement it e.g. like this:

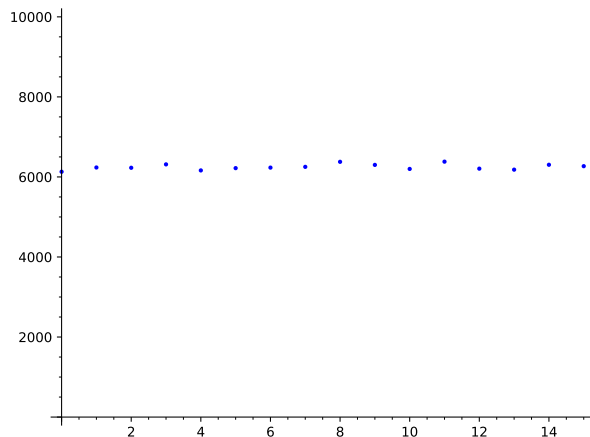
```
1765 sage: import hashlib                                161
1766 sage: def Hash5(x):                                162
1767     ....:     hasher = hashlib.sha256(x)            163
1768     ....:     digest = hasher.hexdigest()           164
1769     ....:     d = ZZ(digest, base=16)               165
1770     ....:     d = d.str(2)[-4:]                     166
1771     ....:     return ZZ(d, base=2)                 167
1772 sage: Hash5(b' ')                                   168
1773 5                                                    169
```

1774 We can then use sage to apply this function to a large set of input values in order to plot a
1775 visualization of the distribution over the set $\{0, \dots, 15\}$. Executing over 500 input values gives:



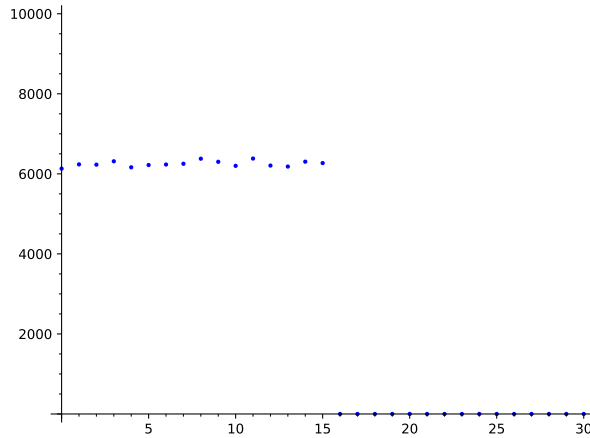
1776

1777 To get an intuition of uniformity, we can count the number of times the hash function $H_{16.nb\text{its}()-1}$
1778 maps onto each number in the set $\{0, 1, \dots, 15\}$ in a loop of 100000 hashes and compare that
1779 to the ideal uniform distribution, which would map exactly 6250 samples to each element. This
1780 gives the following result:



1781

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



The second widely used method of hashing into \mathbb{Z}_r is constructed by observing, that if r is a modulus, with a bit-length of $r.bits() = k_1$ and $H : \{0, 1\}^* \rightarrow \{0, 1\}^{k_2}$ is a hash function that produces digests of size k_2 , with $k_2 \geq k_1$, then a hash-to-group function can be constructed by interpreting the image of H as binary representation of a integer and then take the modulus by r to map into \mathbb{Z}_r . To be more precise

$$H'_{mod_r} : \{0, 1\}^* \rightarrow \mathbb{Z}_r : s \mapsto \left(H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \dots + H(s)_{k_2} \cdot 2^{k_2} \right) \bmod n \quad (4.9)$$

where $H(s)_j$ means the j 's bit of the image binary string $H(s)$ of the original binary hash function.

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

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

$$H_{mod_{23}, k_2} : \{0, 1\}^* \rightarrow \mathbb{Z}_{23} : \\ s \mapsto \left(SHA256(s)_0 \cdot 2^0 + SHA256(s)_1 \cdot 2^1 + \dots + SHA256(s)_{k_2} \cdot 2^{k_2} \right) \bmod 23$$

We want to use various instantiations of k_2 , to visualize the impact of truncation length on the distribution of the hashes in \mathbb{Z}_{23} . We can invoke sage to implement it e.g. like this:

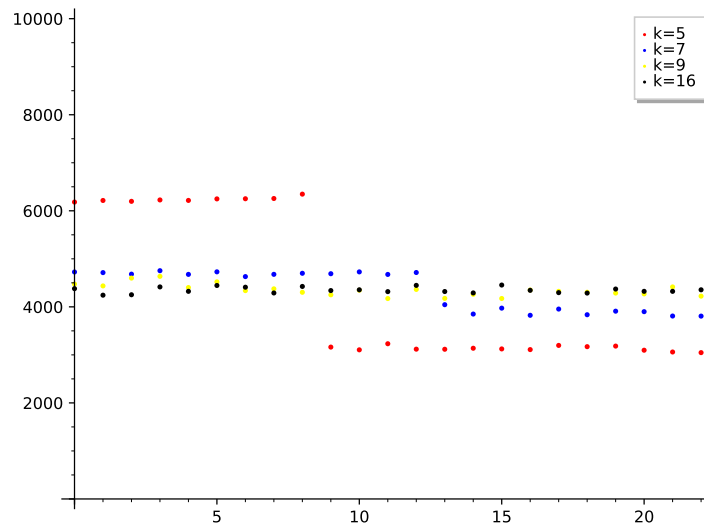
```
sage: import hashlib 170
sage: Z23 = Integers(23) 171
sage: def Hash_mod23(x, k2): 172
.....:     hasher = hashlib.sha256(x.encode('utf-8')) 173
.....:     digest = hasher.hexdigest() 174
```

```

1808 .....:      d = ZZ(digest, base=16)          175
1809 .....:      d = d.str(2)[-k2:]              176
1810 .....:      d = ZZ(d, base=2)                177
1811 .....:      return ZZ3(d)                    178

```

1812 We can then use sage to apply this function to a large set of input values in order to plot visu-
 1813 alizations of the distribution over the set $\{0, \dots, 22\}$ for various values of k_2 by counting the
 1814 number of times it maps onto each number in a loop of 100000 hashes. We get



1815

1816 A third method that can sometimes be found in implementations is the so called *try and*
 1817 *increment method*. To understand this method, we define an integer $z \in \mathbb{Z}$ from any hash value
 1818 $H(s)$ as we did in the previous methods, that is we define $z = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \dots +$
 1819 $H(s)_{k-1} \cdot 2^k$.

1820 Hashing into \mathbb{Z}_r is then achievable by first computing z and then try to see if $z \in \mathbb{Z}_r$. If this
 1821 is the case then the hash is done and if not the string s is modified in a deterministic way and the
 1822 process is repeated until a suitable number z is found. A suitable, deterministic modification
 1823 could be to concatenate the original string by some bit counter. A try and increment algorithm
 would then work like in algorithm XXX

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

Require: $r \in \mathbb{Z}$ with $r.\text{nbits}() = k$ and $s \in \{0, 1\}^*$

procedure TRY-AND-INCREMENT(r, k, s)

$c \leftarrow 0$

repeat

$s' \leftarrow s || c.\text{bits}()$

$z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \dots + H(s')_k \cdot 2^k$

$c \leftarrow c + 1$

until $z < r$

return x

end procedure

Ensure: $z \in \mathbb{Z}_r$

1824

1825 Depending on the parameters, this method can be very efficient. In fact, if k is sufficiently
 1826 large and r is close to 2^{k+1} , the probability for $z < r$ is very high and the repeat loop will almost

always be executed a single time only. A drawback is however that the probability to execute the loop multiple times is not zero.

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

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

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

$$H_{Ped} : (\mathbb{Z}_r)^j \rightarrow \mathbb{G} : (x_1, \dots, x_j) \mapsto \prod_{i=1}^j g_i^{x_i} \quad (4.10)$$

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

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

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

MimC Hashes

Pseudo Random Functions in DDH-A groups As noted in XXX, Pedersen's hash function does not have the properties a random function and should therefore not be instantiated as such. To look at a construction that serves as random oracle function in groups where the decisional Diffie-Hellman construction is assumed to hold true let \mathbb{G} be a DDH-A group of order r with generator g and $\{a_0, a_1, \dots, a_k\} \subset \mathbb{Z}_r^*$ a uniform randomly generated set of numbers invertible in modular r arithmetics. Then a pseudo-random function is given by

$$F_{rand} : \{0, 1\}^{k+1} \rightarrow \mathbb{G} : (b_0, \dots, b_k) \mapsto g^{b_0 \cdot \prod_{i=1}^k a_i^{b_i}} \quad (4.11)$$

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

$$H_{rand, \mathbb{G}} : \{0, 1\}^* \rightarrow \mathbb{G} : s \mapsto F_{rand}(H(s)) \quad (4.12)$$

4.2 Commutative Rings

Thinking of integers again, we know, that there are actually two operations addition and multiplication and as we know addition defines a group structure on the set of integers. However multiplication does not define a group structure as we know that integers in general don't have multiplicative inverses.

Combinations like this are captured by the concept of a so called *commutative ring with unit*. To be more precise, a commutative ring with unit $(R, +, \cdot, 1)$ is a set R , provided with two maps $+: R \cdot R \rightarrow R$ 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 with 0.
- (Commutativity of the multiplication) We have $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.
- (Existence of a unit) There is an element $1 \in R$, such that $1 \cdot g$ holds for all $g \in R$,
- (Associativity) For every $g_1, g_2, g_3 \in \mathbb{G}$ the equation $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$ holds.
- (Distributivity) For all $g_1, g_2, g_3 \in R$ the distributive laws $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

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

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

The following example is more interesting. The motivated reader is encouraged to think through this example, not so much because we need this in what follows, but more so as it helps to detach the reader from familiar styles of computation.

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

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

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

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

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

To see how such a "moonmath equation" can be solved, we have to keep in mind, that rings behaves mostly like normal 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"

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

1883 So even despite this equation looked really alien on the surface, computation was basically
 1884 exactly the way "normal" equation like for fractional numbers are done.

Note however that in a ring, things can be very different, then most are used to, whenever a multiplicative inverse would be needed to solve an equation in the usual way. For example the equation

$$\odot \circ x = \otimes$$

can not be solved for x in the usual way, since there is no multiplicative inverse for \odot in our ring. And in fact looking at the multiplication table we see that no such x exists. On another example the equation

$$\odot \circ x = \odot$$

1885 can has not a single solution but two $x \in \{\star, \otimes\}$. Having no or two solutions is certainly not
 1886 something to expect from types like \mathbb{Q} .

1887 *Example 52.* Considering polynomials again, we note from their definition, that what we have
 1888 called the type R of the coefficients, must in fact be a commutative ring with unit, since we need
 1889 addition, multiplication, commutativity and the existence of a unit for $R[x]$ to have the properties
 1890 we expect.

1891 Now considering R to be a ring, addition and multiplication of polynomials as defined in
 1892 XXX, actually makes $R[x]$ into a commutative ring with unit, too, where the polynomial 1 is the
 1893 multiplicative unit.

1894 *Example 53.* Let n be a modulus and $(\mathbb{Z}_n, +, \cdot)$ the set of all remainder classes of integers
 1895 modulo n , with the projection of integer addition and multiplication as defined in XXX. It can
 1896 be shown that $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring with unit 1.

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

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

This of particular interest in cryptographic and snarks, as it allows for the evaluation of polynomials with coefficients in \mathbb{Z}_n to be evaluated "in the exponent". To be more precise let $p \in \mathbb{Z}_n[x]$

be a polynomial with $p(x) = a_m \cdot x^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0$. Then the previously defined exponential laws XXX imply that

$$\begin{aligned} g^{p(x)} &= g^{a_m \cdot x^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0} \\ &= (g^{x^m})^{a_m} \cdot (g^{x^{m-1}})^{a_{m-1}} \cdot \dots \cdot (g^x)^{a_1} \cdot g^{a_0} \end{aligned}$$

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

As we will see this is a key insight to understand many snark protocols like e.g. Groth16 or XXX.

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

$$3^{(\cdot)} : \mathbb{Z}_4 \rightarrow \mathbb{F}_5^* \quad x \mapsto 3^x$$

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

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

and on the other hand we can use the right hand side of equation to evaluate p at 2 in the exponent of 3, which gives:

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

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

4.3 Fields

In this chapter we started with the definition of a group, which we then expended into the definition of a commutative ring with unit. Those rings generalize the behaviour of integers. In this section we will look at the special case of commutative rings, where every element, other

than the neutral element of addition, has a multiplicative inverse. Those structures behave very much like the rational numbers \mathbb{Q} , which are in a sense an extension of the ring of integers, that is constructed by just including newly defined multiplicative inverses (the fractions) to the integers.

Now considering the definition of a ring XXX again, we define a **field** $(\mathbb{F}, +, \cdot)$ to be a set \mathbb{F} , together 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 law $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

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

The **characteristic** $\text{char}(\mathbb{F})$ 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 to have a *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.

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

```
sage: QQ                                     179
Rational Field                               180
sage: QQ(1/5) # Get an element from the field of rational 181
numbers
1/5                                           182
sage: QQ(1/5) / QQ(3) # Division             183
1/15                                         184
```

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

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

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

Since $1 + 1 = 0$ in the field \mathbb{F}_2 , we know that the characteristic of \mathbb{F}_2 is there, that is we have $\text{char}(\mathbb{F}_2) = 2$.

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

```

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

Exercise 30. Consider our remainder class ring $(\mathbb{F}_5, +, \cdot)$ and show that it is a field. What is the characteristic of \mathbb{F}_5 ?

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

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

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

Prime fields are the foundation for many of the contemporary algebra based cryptographic systems, as they have many desirable properties. One of them is, that since these sets are finite and a prime field of characteristic p can be represented on a computer in roughly $\log_2(p)$ amount of space, no precision problems occur, that are for example unavoidable for computer representations of rational numbers or even the integers, because those sets are infinite.

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

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

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

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

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

A first thing to note is that since \mathbb{F}_5 is a field all rules like bracketing (distributivity), summing ect. are identical to the rules we learned in school when we where dealing with rational, real or complex numbers. We get

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

1991 In this computation we computed the multiplicative inverse of 3 using the identity $x^{-1} = x^{p-2}$ in
 1992 a prime field. This impractical for large prime numbers. Recall that another way of computing
 1993 the multiplicative inverse is the Extended Euclidean algorithm. To see that again, the task is to
 1994 compute $x^{-1} \cdot 3 + t \cdot 5 = 1$, but t is actually irrelevant. We get

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

1996 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 .

1997 **Square Roots** In this part we deal with square numbers also called *quadratic residues* and
 1998 *square roots* in prime fields. This is of particular importance in our studies on elliptic curves as
 1999 only square numbers can actually be points on an elliptic curve.

2000 To make the intuition of quadratic residues and roots precise, let $p \in \mathbb{P}$ be a prime number
 2001 and \mathbb{F}_p its associate prime field. Then a number $x \in \mathbb{F}_p$ is called a **square root** of another
 2002 number $y \in \mathbb{F}_p$, if x is a solution to the equation

$$x^2 = y \quad (4.13)$$

2003 In this case y is called a **quadratic residue**. On the other hand, if y is given and the quadratic
 2004 equation has no x solution, we call y as **quadratic non-residue**. For any $y \in \mathbb{F}_p$ we write

$$\sqrt{y} := \{x \in \mathbb{F}_p \mid x^2 = y\} \quad (4.14)$$

2005 for the set of all square roots of y in the prime field \mathbb{F}_n . (If y is a quadratic non-residue, then
 2006 $\sqrt{y} = \emptyset$ and if $y = 0$, then $\sqrt{y} = \{0\}$)

2007 So roughly speaking, quadratic residues are numbers such that we can take the square root
 2008 from them and quadratic non-residues are numbers that don't have square roots. The situation
 2009 therefore parallels the know case of integers, where some integers like 4 or 9 have square roots
 2010 and others like 2 or 3 don't (as integers).

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

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

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

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

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \text{if } y \text{ has square roots} \\ -1 & \text{if } y \text{ has no square roots} \\ 0 & \text{if } y = 0 \end{cases} \quad (4.15)$$

Example 61. Look at the quadratic residues and non residues in \mathbb{F}_5 from example XXX again, we can deduce the following Legendre symbols, from example XXX.

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

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

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}}. \quad (4.16)$$

Example 62. Look at the quadratic residues and non residues in \mathbb{F}_5 from example XXX again, we can compute the following Legendre symbols using the Euler criterium:

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

Exercise 31. Consider the prime field \mathbb{F}_{13} . Find the set of all pairs $(x, y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ that satisfy the equation

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

2030 **Exponentiation** TO APPEAR...

2031 **Hashing into Prime fields** An important problem in snark development is the ability to hash
 2032 to (various subsets) of elliptic curves. As we will see in XXX those curves are often defined
 2033 over prime fields and hashing to a curve then might start with hashing to the prime field. It is
 2034 therefore of importance to understand who to hash into prime fields.

2035 To understand it, note that in XXX we have looked at a few constructions of how to hash
 2036 into the residue class rings \mathbb{Z}_n for arbitrary $n > 1$. As prime fields are just special instances of
 2037 those rings, all hashing into \mathbb{Z}_n functions can be used for hashing into prime fields, too.

2038 **Extension Fields** We defined prime fields in the previous section. They are the basic building
 2039 blocks for cryptography in general and snarks in particular.

2040 However as we will see in XX so called *pairing based* snark systems are crucially dependent
 2041 on group pairings XXX defined over the group of rational points of elliptic curves. For those
 2042 pairings to be non-trivial the elliptic curve must not only be defined over a prime field but over
 2043 a so called *extension field* of a given prime field.

2044 We therefore have to understand field extensions. To understand them first observe the field
 2045 \mathbb{F}' is called an *extension* of a field \mathbb{F} , if \mathbb{F} is a subfield of \mathbb{F}' , that is \mathbb{F} is a subset of \mathbb{F}' and
 2046 restricting the addition and multiplication laws of \mathbb{F}' to the subset \mathbb{F} recovers the appropriate
 2047 laws of \mathbb{F} .

2048 Now it can be shown, that whenever $p \in \mathbb{P}$ is a prime and $m \in \mathbb{N}$ a natural number, then
 2049 there is a field \mathbb{F}_{p^m} with characteristic p and p^m elements, such that \mathbb{F}_{p^m} is an extension field of
 2050 the prime field \mathbb{F}_p .

2051 Similar to how prime fields \mathbb{F}_p are generated by starting with the ring of integers and then
 2052 divide by a prime number p and keep the remainder, prime field extensions \mathbb{F}_{p^m} are generated by
 2053 starting with the ring $\mathbb{F}_p[x]$ of polynomials and then divide them by an irreducible polynomial
 2054 of degree m and keep the remainder.

2055 To be more precise let $P \in \mathbb{F}_p[x]$ be an irreducible polynomial of degree m with coefficients
 2056 from the given prime field \mathbb{F}_p . Then the underlying set \mathbb{F}_{p^m} of the extension field is given by
 2057 the set of all polynomials with a degree less than m :

$$\mathbb{F}_{p^m} := \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_1x + a_0 \mid a_i \in \mathbb{F}_p\} \quad (4.17)$$

2058 which can be shown to be the set of all remainders when dividing any polynomial $Q \in \mathbb{F}_p[x]$ by
 2059 P . So elements of the extension field are polynomials of degree less than m . This is analog to
 2060 how \mathbb{F}_p is the set of all remainders, when dividing integers by p .

2061 Addition in then inheritec from $\mathbb{F}_p[x]$, which means that addition on \mathbb{F}_{p^m} is defined as normal
 2062 addition of polynomials. To be more precise, we have

$$+ : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \rightarrow \mathbb{F}_{p^m}, \left(\sum_{j=0}^m a_j x^j, \sum_{j=0}^m b_j x^j \right) \mapsto \sum_{j=0}^m (a_j + b_j) x^j \quad (4.18)$$

2063 and we can see that the neutral element is (the polynomial) 0 and that the additive inverse is
 2064 given by the polynomial with all negative coefficients.

2065 Multiplication in inheritec from $\mathbb{F}_p[x]$, too, but we have to divide the result by our modulus
 2066 polynomial P , whenever the degree of the resulting polynomial is equal or greater to m . To be
 2067 more precise, we have

$$\cdot : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \rightarrow \mathbb{F}_{p^m}, \left(\sum_{j=0}^m a_j x^j, \sum_{j=0}^m b_j x^j \right) \mapsto \left(\sum_{n=0}^{2m} \sum_{i=0}^n a_i b_{n-i} x^n \right) \bmod P \quad (4.19)$$

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

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

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

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

To summerize we have seen that when a prime field \mathbb{F}_p is given then any field \mathbb{F}_{p^m} constructed in the above manner is a field extension of \mathbb{F}_p . To be more general a field $\mathbb{F}_{p^{m_2}}$ is a field extension of a field $\mathbb{F}_{p^{m_1}}$, if and only if m_1 divides m_2 and from this we can deduces, that for any given fixed prime number, there are nested sequences of fields

$$\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}} \quad (4.20)$$

whenever the power m_j divides the power m_{j+1} , such that $\mathbb{F}_{p^{m_j}}$ is a subfield of $\mathbb{F}_{p^{m_{j+1}}}$.

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

Example 63 (The Extension field \mathbb{F}_{3^2}). In (XXX) we have constructed the prime field \mathbb{F}_3 . In this example we apply the definition (XXX) of a field extension to construct \mathbb{F}_{3^2} . We start by choosing an irreducible polynomial of degree 2 with coefficients in \mathbb{F}_3 . We try $P(t) = t^2 + 1$. 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

$$P(0) = 0^2 + 1 = 1$$

$$P(1) = 1^2 + 1 = 2$$

$$P(2) = 2^2 + 1 = 1 + 1 = 2$$

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

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

So our extension field contains 9 elements as expected. Addition is defined as addition of polynomials. For example $(t+2) + (2t+2) = (1+2)t + (2+2) = 1$. Doing this computation for all elements 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

2093

2094 As we can see, the group $(\mathbb{F}_3, +)$ is a subgroup of the group $(\mathbb{F}_{3^2}, +)$, obtained by only consid-
 2095 ering the first three rows and columns of this table.

2096 As it was the case in previous examples, we can use the table to deduce the negative of any
 2097 element from \mathbb{F}_{3^2} . For example in \mathbb{F}_{3^2} we have $-(2t+1) = t+2$, since $(2t+1) + (t+2) = 0$

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

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

2098 So the product of $t+2$ and $2t+2$ in \mathbb{F}_{3^2} is 2. Doing this computation for all elements give the
 2099 following multiplication table:

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

2100

2101 As it was the case in previous examples, we can use the table to deduce the multiplicative
 2102 inverse of any non-zero element from \mathbb{F}_{3^2} . For example in \mathbb{F}_{3^2} we have $(2t+1)^{-1} = 2t+2$,
 2103 since $(2t+1) \cdot (2t+2) = 1$.

2104 From the multiplication table we can also see, that the only quadratic residues in \mathbb{F}_{3^2} are
 2105 the set $\{0, 1, 2, t, 2t\}$, with $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1, 2\}$, $\sqrt{2} = \{t, 2t\}$, $\sqrt{t} = \{t+2, 2t+1\}$ and
 2106 $\sqrt{2t} = \{t+1, 2t+2\}$.

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

2. So we compute:

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

2107 Computations in extension fields are arguably on the edge of what can reasonably be done with
 2108 pen and paper. Fortunately sage provides us with a simple way to do the computations.

```

2109 sage: Z3 = GF(3) # prime field 192
2110 sage: Z3t.<t> = Z3[] # polynomials over Z3 193
2111 sage: P = Z3t(t^2+1) 194
2112 sage: P.is_irreducible() 195
2113 True 196
2114 sage: F3_2.<t> = GF(3^2, name='t', modulus=P) 197
2115 sage: F3_2 198
2116 Finite Field in t of size 3^2 199
2117 sage: F3_2(t+2)*F3_2(2*t+2) == F3_2(2) 200
2118 True 201
2119 sage: F3_2(2*t+2)^(-1) # multiplicative inverse 202
2120 2*t + 1 203
2121 sage: # verify our solution to (t+1)(x^2 + (2t+2)) = 2 204
2122 sage: F3_2(t+1)*(F3_2(t)**2 + F3_2(2*t+2)) == F3_2(2) 205
2123 True 206
2124 sage: F3_2(t+1)*(F3_2(2*t)**2 + F3_2(2*t+2)) == F3_2(2) 207
2125 True 208

```

Exercise 32. Consider the extension field \mathbb{F}_{3^2} from the previous example and find all pairs of elements $(x, y) \in \mathbb{F}_{3^2}$, such that

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

Exercise 33. Show that the polynomial $P = x^3 + x + 1$ from $\mathbb{F}_5[x]$ is irreducible. Then consider the extension field \mathbb{F}_{5^3} defined relative to P . Compute the multiplicative inverse of $(2t^2 + 4) \in \mathbb{F}_{5^3}$ using the extended Euklidean algorithm. Then find all $x \in \mathbb{F}_{5^3}$ that solve the equation

$$(2t^2 + 4)(x - (t^2 + 4t + 2)) = (2t + 3)$$

2126 **Hashing into extension fields** In XXX we have seen how to hash into prime fields. As ele-
 2127 ments of extension fields can be seen as polynomials over prime fields, hashing into extension
 2128 fields is therefore possible, if every coefficient of the polynimial is hashed independently.

4.4 Projective Planes

Projective planes are a certain type of geometry defined over some given field, that in a sense extend the concept of the ordinary Euclidean plane by including "points at infinity".

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

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

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

$$[X : Y : Z] := \{(k \cdot X, k \cdot Y, k \cdot Z) \mid k \in \mathbb{F}\} \quad (4.21)$$

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

$$\mathbb{FP}^2 := \{[X : Y : Z] \mid (X, Y, Z) \in \mathbb{F}^3 \text{ with } (X, Y, Z) \neq (0, 0, 0)\} \quad (4.22)$$

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

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

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

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

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

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

To find $\mathbb{F}_3\mathbb{P}^2$, we have to compute the set of all lines in $\mathbb{F}_3 \times \mathbb{F}_3 \times \mathbb{F}_3$ that intersect $(0, 0, 0)$.

Since those lines are parameterized by tuples (x_1, x_2, x_3) . We compute:

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

Those lines define the 13 points in the projective plane $\mathbb{F}_3\mathbb{P}$ as follows

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

2168 This projective plane contains 9 affine points, three points at infinity and one line at infinity.

2169 To understand the ambiguity in projective coordinates a bit better, let's consider the point
 2170 $[1 : 2 : 2]$. As this point in the projective plane is a line in \mathbb{F}_3^3 , it has the projective coordinates
 2171 $(1, 2, 2)$ as well as $(2, 1, 1)$, since the former coordinate give the latter, when multiplied in \mathbb{F}_3 by
 2172 the factor 2. In addition note, that for the same reasons the points $[1 : 2 : 2]$ and $[2 : 1 : 1]$ are the
 2173 same, since their underlying sets are equal.

2174 *Exercise 34.* Construct the so called *Fano plane*, that is the projective plane over the finite field
 2175 \mathbb{F}_2 .

Chapter 5

Elliptic Curves

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

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

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

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

5.1 Elliptic Curve Arithmetics

5.1.1 Short Weierstraß Curves

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

We start with their reprrsentation in affine space. This representation has the advantage that affine points are just pairs of numbers which is more convinient to work with for the beginner. However it has the disadvantage that a special "point at infinity" that is not a point on the curve, is necessary to describe the group structure. We introduce the elliptic curve group law and describe elliptic curve scalar multiplication, which is nothing but an instatation of the exponential map from general cyclic groups.

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

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

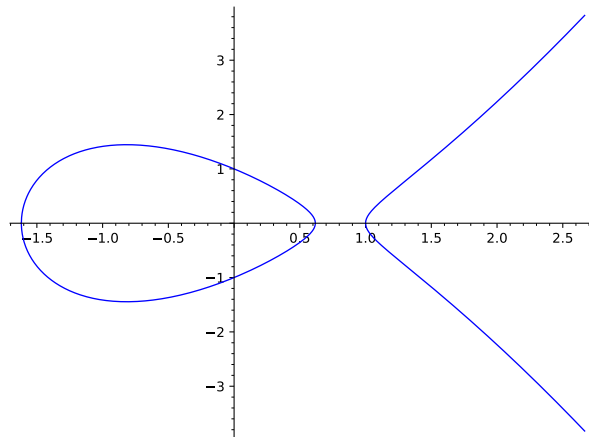
Affine short Weierstraß form Probably the least abstract and most straightforward way to introduce elliptic curves for non-mathematicians and beginners is the so-called affine representation of a short Weierstraß curve. To see what this is, let \mathbb{F} be a finite field of order q and $a, b \in \mathbb{F}$ two field elements such that $4a^3 + 27b^2 \bmod q \neq 0$. Then a **short Weierstrass elliptic curve** $E(\mathbb{F})$ over \mathbb{F} in its affine representation is the set

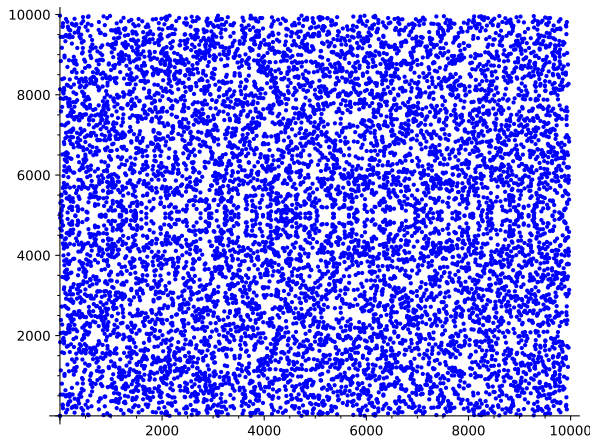
$$E(\mathbb{F}) = \{(x, y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + a \cdot x + b\} \cup \{\mathcal{O}\} \quad (5.1)$$

of all pairs of field elements $(x, y) \in \mathbb{F} \times \mathbb{F}$, that satisfy the short Weierstrass cubic equation $y^2 = x^3 + a \cdot x + b$, together with a distinguished symbol \mathcal{O} , called the **point at infinity**.

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

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





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

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

When dealing with elliptic curves computations can quickly become cumbersome and tedious. So on the one hand side the reader is advised to do as many computations in a pen and paper style as possible. This helps a lot to get a deeper understanding for the details. On the other hand side however, computations are sometimes simply too large to be done by hand and one might get lost in the details. Fortunately sage is very helpful in dealing with elliptic curves. It is there a goal of this book to introduce the reader to the great elliptic curve capabilities of sage. One way to define elliptic curves and work with them goes like this:

```

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

```

The following three examples will give a more practical understanding of what an elliptic curve is and how we can compute them. The reader is advised to read them carefully and ideally to parallel the computation themselves. We will repeatedly build on these examples in this chapter

and use the second example at various places in this book.

Example 65. To provide the reader with a small example of an elliptic curve, where all computation can be done in a pen and paper style, consider the prime field \mathbb{F}_5 from example (XXX). The reader who had worked through the examples and exercises in the previous section knows this prime field well.

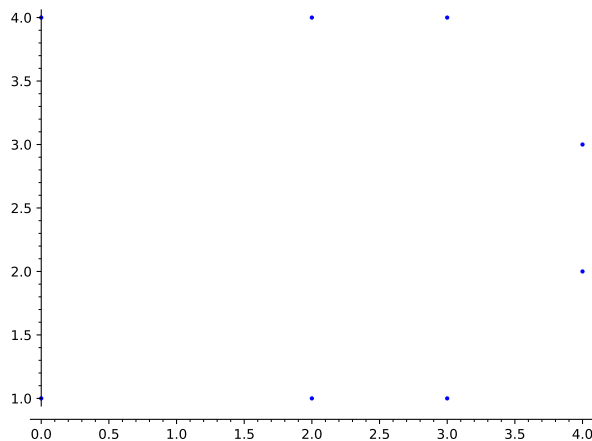
To define an elliptic curve over \mathbb{F}_5 , we have to choose two numbers a and b from that field. Assuming we choose $a = 1$ and $b = 1$ then $4a^3 + 27b^2 \equiv 1 \pmod{5}$ from which follows that the corresponding elliptic curve $E_1(\mathbb{F}_5)$ is given by the set of all pairs (x, y) from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

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

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

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

So our elliptic curve is a set of 9 elements. 8 of which are pairs of numbers and one special symbol \mathcal{O} . Visualizing E_1 gives:



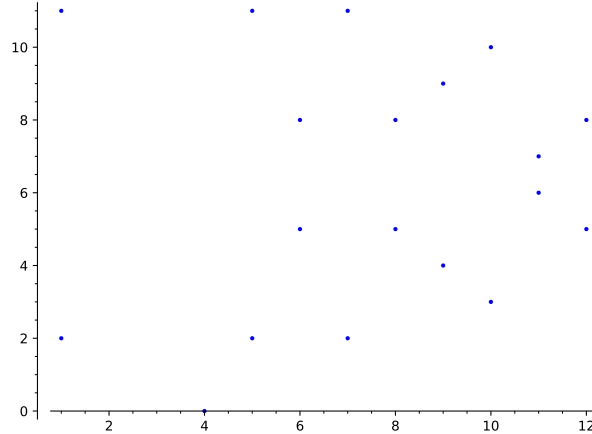
In the development of SNARKS it is sometimes necessary to do elliptic curve cryptography "in a circuit", which basically means that the elliptic curves need to be implemented in a certain SNARK-friendly way. We will look at what this means in XXX. To be able to do this efficiently it is desirable to have curves with special properties. The following example is a pen and paper version of such a curve, that parallels the definition of a cryptographically secure curve called *Baby-JubJub* which is extensively used in real world snarks. The interested reader is advised to read this example carefully as we will use it and build on it in various places throughout the book.

Example 66 (Pen-JubJub). Consider the prime field \mathbb{F}_{13} from exercise XXX. If we choose $a = 8$ and $b = 8$ then $4a^3 + 27b^2 \equiv 6 \pmod{13}$ and the corresponding elliptic curve is given by all pairs (x, y) from \mathbb{F}_{13} such that $y^2 = x^3 + 8x + 8$ holds. We write PJJ_{13} for this curve and call it the *Pen-JubJub* curve.

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

$$PJJ_13 = \{\emptyset, (1, 2), (1, 11), (4, 0), (5, 2), (5, 11), (6, 5), (6, 8), (7, 2), (7, 11), (8, 5), (8, 8), (9, 4), (9, 9), (10, 3), (10, 10), (11, 6), (11, 7), (12, 5), (12, 8)\}$$

As we can see the curve consist of 20 points. 19 points from the affine plane and the point at infinity. To get a visual impression of the *PJJ_13* curve, we might plot all of its points (except the point at infinity) in the $\mathbb{F}_{13} \times \mathbb{F}_{13}$ affine plane. We get:



As we will see in what follows this curve is kind of special as it is possible to represent it in two alternative forms, called the Montgomery and the twisted Edwards form (See xxx and XXX).

Now that we have seen two pen and paper friendly elliptic curves, lets look at a curve that is used in actual cryptography. Cryptographically secure elliptic curve are not qualitatively defferent from the curves we looked at so far. The only difference is that the prime number modulus of the prime field is much larger. Typical examples use prime numbers, which have binary representations in the size of more then double the size of the desired security level. So if for example a security of 128 bit is desired, a prime moduls of binary size ≥ 256 is choosen. The following example provides such a curve.

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

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

which has a binary representation that need 256 bits. This implies that the \mathbb{F}_p approximately contains 2^{256} many elements. So the underlying field is large. To get an image of how large the base field is, consider that the number 2^{256} is approximately in the same order of magnitute as the estimated number of atomes in the observeable universe.

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

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

Clearly Secp256k1 is a curve, to large to do computations by hand, since it can be shown that *Secp256k1* contains

$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

many elements, where r is a prime number that also has a binary representation of 256 bits. Cryptographically secure elliptic curves are therefore not useful in pen and paper computations. Fortunately sage handles large curve efficiently:

```

sage: p = 1157920892373161954235709850086879078532699846656405 226
      64039457584007908834671663
sage: # Hexadecimal representation 227
sage: p.str(16) 228
fffffffffffefffffc 229
      2f
sage: p.is_prime() 230
True 231
sage: p.nbits() 232
256 233
sage: Fp = GF(p) 234
sage: Secp256k1 = EllipticCurve(Fp, [0, 7]) 235
sage: r = Secp256k1.order() # number of elements 236
sage: r.str(16) 237
fffffffffffebaaedce6af48a03bbfd25e8cd03641 238
      41
sage: r.is_prime() 239
True 240
sage: r.nbits() 241
256 242

```

Exercise 35. Look-up the definition of curve BLS12-381, implement it in sage and compute its order.

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

To reduce the amount of space needed to represent a curve point note however, that up to a sign the y -coordinate of a curve point can be computed from the x -coordinate, by simply inserting x into the Weierstrass equation and then computing the roots of the result. This gives two results and it follows that we can represent a curve point in **compressed form** by simply storing the x -coordinate together with a single sign bit only, the latter of which deterministically decides which of the two roots to choose. In case that the y -coordinate is zero, both sign bits give the same result.

For example one convention could be to always choose the root closer to 0, when the sign bit is 0 and the root closer to the order of \mathbb{F} when the sign bit is 1.

Example 68 (Pen-JubJub). To understand the concept of compressed curve points a bit better consider the *PJJ_13* curve from example XXX again. Since this curve is defined over the prime field \mathbb{F}_{13} and numbers between 0 and 13 need approximately 4 bits to be represented, each *PJJ_13*-point needs 8-bits of storage in uncompressed form, while it would need only 5 bits in compressed form. To see how this works, recall that in uncompressed form we have

$$PJJ_{13} = \{\mathcal{O}, (1, 2), (1, 11), (4, 0), (5, 2), (5, 11), (6, 5), (6, 8), (7, 2), (7, 11), (8, 5), (8, 8), (9, 4), (9, 9), (10, 3), (10, 10), (11, 6), (11, 7), (12, 5), (12, 8)\}$$

Using the technique of point compression, we can replace the y -coordinate in each (x, y) pair by a sign bit, indicating, whether or not y is closer to 0 or to 13. So y values in the range $[0, \dots, 6]$ having sign bit 0 and y -values in the range $[7, \dots, 12]$ having sign bit 1. Applying this to the points in PJJ_13 gives the compressed representation:

$$PJJ_13 = \{\mathcal{O}, (1, 0), (1, 1), (4, 0), (5, 0), (5, 1), (6, 0), (6, 1), (7, 0), (7, 1), (8, 0), (8, 1), (9, 0), (9, 1), (10, 0), (10, 1), (11, 0), (11, 1), (12, 0), (12, 1)\}$$

2354 Note that the numbers $7, \dots, 12$ are the negatives (additive inverses) of the numbers $1, \dots, 6$ in
 2355 modular 13 arithmetics and that $-0 = 0$. Calling the compression bit a "sign bit" therefore
 2356 makes sense.

2357 To recover the uncompressed point of say $(5, 1)$, we insert the x -coordinate 5 into the Weier-
 2358 straß equation and get $y^2 = 5^3 + 8 \cdot 5 + 8 = 4$. As expected 4 is a quadratic residue in \mathbb{F}_{13} with
 2359 roots $\sqrt{4} = \{2, 11\}$. Now since the sign bit of the point is 1, we have to choose the root closer
 2360 to the modulus 13 which is 11. The uncompressed point is therefore $(5, 11)$.

2361 Looking at the previous examples, compression rate looks not very impressive. The followin
 2362 example therefore looks at the Secp256k1 curve to show that compression is actually useful.

2363 *Example 69.* Consider the Secp256k1 curve from example XXX again. The following code
 2364 involves sage to generate a random affine curve point, we then apply our compression method

```

2365 sage: P = Secp256k1.random_point().xy()                                243
2366 sage: P                                                                244
2367 (5583057615077650162744874925391800483014826801296554289265412 245
2368    2424383315402997, 17740683673296944980181113999760415857439
2369    203922130191180403105297080125716668)
2370 sage: # uncompressed affine point size                                246
2371 sage: ZZ(P[0]).nbits()+ZZ(P[1]).nbits()                               247
2372 509                                                                    248
2373 sage: # compute the compression                                       249
2374 sage: if P[1] > Fp(-1)/Fp(2):                                         250
2375     ....:     PARITY = 1                                              251
2376     ....: else:                                                        252
2377     ....:     PARITY = 0                                              253
2378 sage: PCOMPRESSED = [P[0], PARITY]                                    254
2379 sage: PCOMPRESSED                                                     255
2380 [5583057615077650162744874925391800483014826801296554289265412 256
2381    2424383315402997, 0]
2382 sage: # compressed affine point size                                   257
2383 sage: ZZ(PCOMPRESSED[0]).nbits()+ZZ(PCOMPRESSED[1]).nbits()          258
2384 255                                                                    259

```

2385 **Affine group law** One of the key properties of an elliptic curve is that it is possible to define
 2386 a group law on the set of its rational points, such that the point at infinity serves as the neutral
 2387 element and inverses are reflections on the x -axis.

2388 The origin of this law can be understood in a geometric picture and is known as the *chord-*
 2389 *and-tangent rule*. In the affine representation of a short Weierstraß curve, the rule can be de-
 2390 scribed in the following way:

- (Point addition) Let $P, Q \in E(\mathbb{F}) \setminus \{\mathcal{O}\}$ with $P \neq Q$ be two distinct points on an elliptic curve, that are both not the point at infinity. Then the sum of P and Q is defined as follows: Consider the line l which intersects the curve in P and Q . If l intersects the elliptic curve at a third point R' , define the sum $R = P \oplus Q$ of P and Q as the reflection of R' at the x -axis. If it does not intersect the curve at a third point define the sum to be the point at infinity \mathcal{O} . It can be shown, that no such chord-line will intersect the curve in more then three points, so addition is not ambiguous.
- (Point doubling) Let $P \in E(\mathbb{F}) \setminus \{\mathcal{O}\}$ be a point on an elliptic curve, that is not the point at infinity. Then the sum of P with itself (the doubling) is defined as follows: Consider the line wich is tangent to the elliptic curve at P , if this line intersects the elliptic curve at a second point R' . The sum $2P = P + P$ is then the reflection of R' at the x -axis. If it does not intersect the curve at a third point define the sum to be the point at infinity \mathcal{O} . It can be shown, It can be shown, that no such tangent-line will intersect the curve in more then two points, so addition is not ambiguous.
- (Point at infinity) We define the point at infinity \mathcal{O} as the neutral ement of addition, that is we define $P + \mathcal{O} = P$ for all points $P \in E(\mathbb{F})$.

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

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

- (Neutral element) Point at infinity \mathcal{O} is the neutral element.
- (Additive inverse) The additive inverse of \mathcal{O} is \mathcal{O} and for any other curve point $(x, y) \in E(\mathbb{F}) \setminus \{\mathcal{O}\}$, the additive inverse is given by $(x, -y)$.
- (Addition rule) For any two curve points $P, Q \in E(\mathbb{F})$ addition is defined by one of the following three cases:
 1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P \oplus Q = P$.
 2. (Adding inverse elements) If $P = (x, y)$ and $Q = (x, -y)$, the sum is defined as $P \oplus Q = \mathcal{O}$.
 3. (Adding non self-inverse equal points) If $P = (x, y)$ and $Q = (x, y)$ with $y \neq 0$, the sum $2P = (x', y')$ is defined by

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

4. (Adding non inverse differen points) If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $x_1 \neq x_2$, the sum $R = P + Q$ with $R = (x_3, y_3)$ is defined by

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

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

2424 *Notation and Symbols* 8. Let \mathbb{F} be a field and $E(\mathbb{F})$ be an elliptic curve over \mathbb{F} . We write \oplus for
 2425 the group law on $E(\mathbb{F})$ and $(E(\mathbb{F}), \oplus)$ for the group of rational points.

2426 As we can see, it is very efficient to compute inverses on elliptic curves. However computing
 2427 the addition of elliptic curve points in the affine representation needs to consider many cases
 2428 and involves extensive finite field divisions. As we will see in the next paragraph this can be
 2429 simplified in projective coordinates.

2430 To get some practical impression of how the group law on an elliptic curve is computed, lets
 2431 look at some actual cases:

Example 70. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX again. As we have seen, the set of rational points contains 9 elements and is given by

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

2432 We know that this set defines a group, so we can add any two elements from $E_1(\mathbb{F}_5)$ to get a
 2433 third element.

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

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

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

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

get

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

2434 So in our elliptic curve $E_1(\mathbb{F}_5)$ we get the doubling $2 \cdot (4, 2)$, that is $(4, 2) \oplus (4, 2) = (3, 4)$ and
 2435 indeed the pair $(3, 4)$ is an element of $E_1(\mathbb{F}_5)$ as expected. The group $E_1(\mathbb{F}_5)$ has no self inverse
 2436 points other than the neutral element \mathcal{O} , since no point has 0 as its y-coordinate. We can invoke
 2437 sage to double check the computations.

```

2438 sage: F5 = GF(5)                                260
2439 sage: E1 = EllipticCurve(F5, [1, 1])              261
2440 sage: INF = E1(0) # point at infinity              262
2441 sage: P1 = E1(0, 1)                                263
2442 sage: P2 = E1(4, 2)                                264
2443 sage: P3 = E1(0, 4)                                265
2444 sage: R1 = E1(2, 1)                                266
2445 sage: R2 = E1(3, 4)                                267
2446 sage: R1 == P1+P2                                  268
2447 True                                              269
2448 sage: INF == P1+P3                                  270
2449 True                                              271
2450 sage: R2 == P2+P2                                  272
2451 True                                              273
2452 sage: R2 == 2*P2                                    274
2453 True                                              275
2454 sage: P3 == P3 + INF                                276
2455 True                                              277

```

Example 71 (Pen-JubJub). Consider the *PJJ_13*-curve from example XXX again and recall that its group of rational points is given by

$$\begin{aligned}
 PJJ_13 = \{ & \mathcal{O}, (1, 2), (1, 11), (4, 0), (5, 2), (5, 11), (6, 5), (6, 8), (7, 2), (7, 11), \\
 & (8, 5), (8, 8), (9, 4), (9, 9), (10, 3), (10, 10), (11, 6), (11, 7), (12, 5), (12, 8) \}
 \end{aligned}$$

2456 In contrast to the group from the previous example, this group contains a self inverse point,
 2457 which is different from the neutral element, given by $(4, 0)$. To see what this means, observe

that we can not add $(4,0)$ to itself using the tangent rule 3 from XXX, as the y-coordinate is zero. Instead we have to use rule 2, since $0 = -0$. We therefore get $(4,0) \oplus (4,0) = \mathcal{O}$ in *PJJ_13*. The point $(4,0)$ is therefore inverse to itself, as adding it to itself gives the neutral element.

```

2462 sage: F13 = GF(13)                                278
2463 sage: MJJ = EllipticCurve(F13, [8, 8])              279
2464 sage: P = MJJ(4, 0)                                  280
2465 sage: INF = MJJ(0) # Point at infinity              281
2466 sage: INF == P+P                                     282
2467 True                                                283
2468 sage: INF == 2*P                                     284
2469 True                                                285

```

Example 72. Consider the Secp256k1 curve from example XXX again. The following code involves sage to generate a random affine curve point, we then apply our compression method

```

2472 sage: P = Secp256k1.random_point()                  286
2473 sage: Q = Secp256k1.random_point()                  287
2474 sage: INF = Secp256k1(0)                            288
2475 sage: R1 = -P                                       289
2476 sage: R2 = P + Q                                   290
2477 sage: R3 = Secp256k1.order()*P                     291
2478 sage: P.xy()                                       292
2479 (4839385178602455275019826924488914207304740149544864010102436 293
2480    1457140422872652, 88911941591273472093376821517671417971298
2481    658884711211512273048237659775389159)
2482 sage: Q.xy()                                       294
2483 (6829518048757273021112907145307122863157071565926748466711313 295
2484    6645704759157781, 37052921081256915815118577727773384838371
2485    706865345203605351471953607504111055)
2486 sage: (ZZ(R1[0]).str(16), ZZ(R1[1]).str(16))        296
2487 ('6afdf30f00747919b2f6b3ea53fb958060ba153731a30dfc449d2cf76eef 297
2488    0e4c', '3b6d9fd9cd064336088df6d59fe1f09e2cac21e22faaf3dd1
2489    aac9d09d3a6ae48')
2490 sage: R2.xy()                                       298
2491 (4747098472388641160824829610831369897726631989885272850477391 299
2492    0418522052665992, 75709654328655049441208497149737998128813
2493    391344329486999646993557763069600721)
2494 sage: R3 == INF                                    300
2495 True                                                301
2496 sage: P[1]+R1[1] == Fp(0) # -(x,y) = (x,-y)        302
2497 True                                                303

```

Exercise 36. Consider the *PJJ_13*-curve from example XXX.

- 2499 1. Compute the inverse of $(10,10)$, \mathcal{O} , $(4,0)$ and $(1,2)$.
- 2500 2. Compute the expression $3 * (1,11) - (9,9)$.
- 2501 3. Solve the equation $x + 2(9,4) = (5,2)$ for some $x \in PJJ_13$

4. Solve the equation $x \cdot (7, 11) = (8, 5)$ for $x \in \mathbb{Z}$

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

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

Now, elliptic curve scalar multiplication is then nothing but the exponential map, written in additive notation. To be more precise let \mathbb{F} be a finite field, $E(\mathbb{F})$ an elliptic curve of order r and P a generator of $E(\mathbb{F})$. Then the **elliptic curve scalar multiplication** with base P is given by

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

where $[0]P = \mathcal{O}$ and $[m]P = P + P + \dots + P$ is the m -fold sum of P with itself. Elliptic curve scalar multiplication is therefore nothing but an instantiation of the general exponential map, when using additive instead of multiplicative notation. This map is a homomorphism of groups, which means that $[n + m]P = [n]P \oplus [m]P$.

As with all finite, cyclic groups the inverse of the exponential map exist and is usually called the *elliptic curve discrete logarithm map*. However elliptic curve are believed to be XXX-groups, which means that we don't know of any efficient way to actually compute this map.

Scalar multiplication and its inverse, the elliptic curve discrete logarithm, define the elliptic curve discrete logarithm *problem*, which consists of finding solutions $m \in \mathbb{Z}_r$, such that

$$P = [m]Q \tag{5.2}$$

holds. Any solution m is usually called a *discrete logarithm* relation between P and Q . If Q is a generator of the curve, then there is a discrete logarithm relation between Q and any other point, since Q generates the group by repeatedly adding Q to itself. So for generator Q and point P , we know some discrete logarithm relation exist. However since elliptic curves are believed to be XXX-groups, finding actual relations m is computationally hard, with runtimes approximately in the size of the order of the group. In practice we often need the assumption that a discrete logarithm relation exists, but that at the same time no one knows this relation.

One useful property of the exponential map in regard to the examples in this book, is that it can be used to greatly simplify pen and paper computations. As we have seen in example XXX, computing the elliptic curve addition law takes quite a bit of effort, when done without a computer. However when g is a generator of small pen and paper elliptic curve group of order r , we can use the exponential map to write the group as

$$\mathbb{G} = \{[1]g \rightarrow [2]g \rightarrow [3]g \rightarrow \dots \rightarrow [r-1]g \rightarrow \mathcal{O}\} \tag{5.3}$$

using cofactor clearing, which implies that $[r]g = \mathcal{O}$. "Logarithmic ordering" like this greatly simplifies complicated elliptic curve addition to the much simpler case of modular r addition. So in order to add two curve points P and Q , we only have to look up their discrete log relations with the generator, say $P = [n]g$ and $Q = [m]g$ and compute the sum as $P \oplus Q = [n + m]g$. This is, of course, only possible for small groups which we can organize as in XXX.

In the following example we will look at some implications of the fact that elliptic curves are finite cyclic groups. We will apply the fundamental theorem of finite cyclic groups and look how it reflects on the curves in consideration.

2539 *Example 73.* Consider the elliptic curve group $E_1(\mathbb{F}_5)$ from example XXX. Since it is a finite
 2540 cyclic group of order 9 and the prime factorization of 9 is $3 \cdot 3$, we can use the fundamental
 2541 theorem of finite cyclic groups to reason about all its subgroups. In fact since the only prime
 2542 factor of 9 is 3, we know that $E_1(\mathbb{F}_5)$ has the following subgroups:

- 2543 • $\mathbb{G}_1 = E_1(\mathbb{F}_5)$ is a subgroup of order 9. By definition any group is a subgroup of itself.
- 2544 • $\mathbb{G}_2 = \{(2, 1), (2, 4), \mathcal{O}\}$ is a subgroup of order 3. This is the subgroup associated to the
 2545 prime factor 3.
- 2546 • $\mathbb{G}_3 = \{\mathcal{O}\}$ is a subgroup of order 1. This is the trivial subgroup.

Moreover since $E_1(\mathbb{F}_5)$ and all its subgroups are cyclic, we know from XXX, that they must have generators. For example the curve point $(2, 1)$ is a generator of the order 3-subgroup \mathbb{G}_2 , since every element of \mathbb{G}_2 can be generated, by repeatedly adding $(2, 1)$ to itself:

$$\begin{aligned} [1](2, 1) &= (2, 1) \\ [2](2, 1) &= (2, 4) \\ [3](2, 1) &= \mathcal{O} \end{aligned}$$

Since $(2, 1)$ is a generator we know from XXX, that it gives rise to an exponential map from the finite field \mathbb{F}_3 onto \mathbb{G}_2 defined by scalar multiplication

$$[\cdot](2, 1) : \mathbb{F}_3 \rightarrow \mathbb{G}_2 : x \mapsto [x](2, 1)$$

To give an example of a generator that generates the entire group $E_1(\mathbb{F}_5)$ consider the point $(0, 1)$. Applying the tangent rule repeatedly we compute with some effort:

$$\begin{array}{ll} [0](0, 1) = \mathcal{O} & [1](0, 1) = (0, 1) \\ [2](0, 1) = (4, 2) & [3](0, 1) = (2, 1) \\ [4](0, 1) = (3, 4) & [5](0, 1) = (3, 1) \\ [6](0, 1) = (2, 4) & [7](0, 1) = (4, 3) \\ [8](0, 1) = (0, 4) & [9](0, 1) = \mathcal{O} \end{array}$$

Again, since $(2, 1)$ is a generator we know from XXX, that it gives rise to an exponential map. However since the group order is not a prime number, the exponential maps, does not map a from any field but from the residue class ring \mathbb{Z}_9 only:

$$[\cdot](0, 1) : \mathbb{Z}_9 \rightarrow \mathbb{G}_1 : x \mapsto [x](0, 1)$$

Using the generator $(0, 1)$ and its associated exponential map, we can write $E(\mathbb{F}_1)$ i logarithmic order with respect to $(0, 1)$ as explained in XXX. We get

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

2547 indicating that the first element is a generator and the n -th element is the scalar product of n and
 2548 the generator. To how this logarithmic orders like this simplify the computations in small elliptic
 2549 curve groups, consider example XXX again. In that example we use the chord and tangent rule
 2550 to compute $(0, 1) \oplus (4, 2)$. Now in the logarithmic order of $E_1(\mathbb{F})$ we can compute that sum
 2551 much easier, since we can directly see that $(0, 1) = [1](0, 1)$ and $(4, 2) = [2](0, 1)$. We can then
 2552 deduce $(0, 1) \oplus (4, 2) = (2, 1)$ immediately, since $[1](0, 1) \oplus [2](0, 1) = [3](0, 1) = (2, 1)$.

To give another example, we can immediately see that $(3,4) \oplus (4,3) = (4,2)$, without doing any expensive elliptic curve addition, since we know $(3,4) = [4](0,1)$ as well as $(4,3) = [7](0,1)$ from the logarithmic representation of $E_1(\mathbb{F}_5)$ and since $4+7=2$ in \mathbb{Z}_9 , the result must be $[2](0,1) = (4,2)$.

Finally we can use $E_1(\mathbb{F}_5)$ as an example to understand the concept of cofactor clearing from XXX. Since the order of $E_1(\mathbb{F}_5)$ is 9 we only have a single factor, which happen to be the cofactor as well. Cofactor clearing then implies that we can map any element from $E_1(\mathbb{F}_5)$ onto its prime factor group \mathbb{G}_2 by scalar multiplication with 3. For example taking the element $(3,4)$ which is not in \mathbb{G}_2 and multiplying it with 3, we get $[3](3,4) = (2,1)$, which is an element of \mathbb{G}_2 as expected.

In the following example we will look at the subgroups of our pen-jubjub curve, define generators and compute the logarithmic order for pen and paper computations. Then we have another look at the principle of cofactor clearing.

Example 74. Consider the pen-jubjub curve PJJ_13 from example XXX again. Since the order of PJJ_13 is 20 and the prime factorization of 20 is $2^2 \cdot 5$, we know that the PJJ_13 contains a "large" prime order subgroup of size 5 and a small prime order subgroup of size 2.

To compute those groups we can apply the technique of cofactor clearing in a try and repeat loop. We start the loop by arbitrarily choose an element $P \in PJJ_13$. Then we multiply that element with the cofactor of the group, we want to compute. If the result is \mathcal{O} , we try a different element and repeat the process until the result is different from the point at infinity.

To compute a generator for the small prime order subgroup $(PJJ_13)_2$, first observe that the cofactor is 10, since $20 = 2 \cdot 10$. We then arbitrarily choose the curve point $(5,11) \in PJJ_13$ and compute $[10](5,11) = \mathcal{O}$. Since the result is the point at infinity, we have to try another curve point, say $(9,4)$. We get $[10](9,4) = (4,0)$ and we can deduce that $(4,0)$ is a generator of $(PJJ_13)_2$. Logarithmic order of then gives

$$(PJJ_13)_2 = \{(4,0) \rightarrow \mathcal{O}\}$$

as expected, since we know from example XXX that $(4,0)$ is self inverse, with $(4,0) \oplus (4,0) = \mathcal{O}$. Double checking the computations using sage:

```

sage: F13 = GF(13)                                     304
sage: PJJ = EllipticCurve(F13, [8, 8])                 305
sage: P = PJJ(5, 11)                                   306
sage: INF = PJJ(0)                                     307
sage: 10*P == INF                                       308
True                                                    309
sage: Q = PJJ(9, 4)                                    310
sage: R = PJJ(4, 0)                                    311
sage: 10*Q == R                                         312
True                                                    313

```

We can apply the same reasoning to the "large" prime order subgroup $(PJJ_13)_5$, which contains 5 elements. To compute a generator for this group, first observe that the associated cofactor is 4, since $20 = 5 \cdot 4$. We choose the curve point $(9,4) \in PJJ_13$ again and compute $[4](9,4) = (7,11)$ and we can deduce that $(7,11)$ is a generator of $(PJJ_13)_5$. Using the gener-

ator $(7, 11)$, we compute the exponential map $[\cdot](7, 11) : \mathbb{F}_5 \rightarrow PJJ_13$ and get

$$\begin{aligned} [0](7, 11) &= \mathcal{O} \\ [1](7, 11) &= (7, 11) \\ [2](7, 11) &= (8, 5) \\ [3](7, 11) &= (8, 8) \\ [4](7, 11) &= (7, 2) \end{aligned}$$

We can use this computation to write the large order prime group $(PJJ_13)_5$ of the pen-jubjub curve in logarithmic order, which we will use quite frequently in what follows. We get:

$$(PJJ_13)_5 = \{(7, 11) \rightarrow (8, 5) \rightarrow (8, 8) \rightarrow (7, 2) \rightarrow \mathcal{O}\}$$

2585 From this, we can immediately see that for example $(8, 8) \oplus (7, 2) = (8, 5)$, since $3 + 4 = 2$ in
2586 \mathbb{F}_5 .

2587 From the previous two examples, the reader might get the impression, that elliptic curve
2588 computation can be largely replaced by modular arithmetics. This however is not true in gen-
2589 eral, but only an arefact of small groups where it is possible to write the entire group in a
2590 logarithmic order. The following example gives some understanding, why this is not possible
2591 in cryptographically secure groups

2592 *Example 75.* SEKTP BICOIN. DISCRET LOG HARDNESS PROHIBITS ADDITION IN
2593 THE FIELD...

2594 **Projective short Weierstraß form** As we have seen in the previous section, describing ellip-
2595 tic curves as pairs of points that satisfy a certain equation is relatively straight forward. However
2596 in order to define a group structure on the set of points, we had to add a special point at infinity
2597 to act as the neutral element.

2598 Recalling from the definition of projective planes XXX we know, that points at infinity
2599 are handled as ordinary points in projective geometry. It make therefore sense to look at the
2600 definition of a short Weierstraß curve in projective geometry.

2601 To see what a short Weierstraß curve in projective coordinates is, let \mathbb{F} be a finite field of
2602 order q and characteristic > 3 , $a, b \in \mathbb{F}$ two field elements such that $4a^3 + 27b^2 \bmod q \neq 0$ and
2603 \mathbb{FP}^2 the projective plane over \mathbb{F} . Then a **short Weierstrass elliptic curve** over \mathbb{F} in its projective
2604 representation is the set

$$E(\mathbb{FP}^2) = \{[X : Y : Z] \in \mathbb{FP}^2 \mid Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3\} \quad (5.4)$$

2605 of all points $[X : Y : Z] \in \mathbb{FP}^2$ from the projective plane, that satisfy the *homogenous* cubic
2606 equation $Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3$.

To understand how the point at infinity is unified in this definition, recall from XXX that,
in projective geometry points at infinity are given by homogeneous coordinates $[X : Y : 0]$.
Inserting representatives $(x_1, y_1, 0) \in [X : Y : 0]$ from those classes into the defining homogenous
cubic equations gives

$$\begin{aligned} y_1^2 \cdot 0 &= x_1^3 + a \cdot x_1 \cdot 0^2 + b \cdot 0^3 && \Leftrightarrow \\ 0 &= x_1^3 \end{aligned}$$

which shows that the only point at infinity that is also a point on a projective short Weierstraß
curve is the class

$$[0, 1, 0] = \{(0, y, 0) \mid y \in \mathbb{F}\}$$

2607 This point is the projective representation of \mathcal{O} . The projective representation of a short Weier-
 2608 straß curve therefore has the advantage to not need a special symbol to represent the point at
 2609 infinity \mathcal{O} from the affine definition.

Example 76. To get an intuition of how an elliptic curve in projective geometry looks, consider curve $E_1(\mathbb{F}_5)$ from example (XXX). We know that in its affine representation, the set of rational points is given by

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

2610 which is defined as the set of all pairs $(x, y) \in \mathbb{F}_5 \times \mathbb{F}_5$, such that the affine short Weierstrass
 2611 equation $y^2 = x^3 + ax + b$ with $a = 1$ and $b = 1$ is satisfied.

To find the projective representation of a short Weierstrass curve with the same parameters $a = 1$ and $b = 1$, we have to compute the set of projective points $[X : Y : Z]$ from the projective plane $\mathbb{F}_5\mathbb{P}^2$, that satisfy the homogenous cubic equation

$$y_1^2 z_1 = x_1^3 + 1 \cdot x_1 z_1^2 + 1 \cdot z_1^3$$

2612 for any representative $(x_1, y_1, z_1) \in [X : Y : Z]$. We know from XXX, that the projective plane
 2613 $\mathbb{F}_5\mathbb{P}^2$ contains $5^2 + 5 + 1 = 31$ elements, so we can take the effort and insert all elements into
 2614 equation XXX and see if both sides match.

For example, consider the projective point $[0 : 4 : 1]$. We know from XXX, that this point in the projective plane represents the line

$$[0 : 4 : 1] = \{(0, 0, 0), (0, 4, 1), (0, 3, 2), (0, 2, 3), (0, 1, 4)\}$$

in the three dimensional space \mathbb{F}^3 . To check whether or not $[0 : 4 : 1]$ satisfies XXX, we can insert any representative, that is we can insert any element from XXX. Each element satisfies the equation if and only if any other satisfies the equation. So we insert $(0, 4, 1)$ and get

$$1^2 \cdot 1 = 0^3 + 1 \cdot 0 \cdot 1^2 + 1 \cdot 1^3$$

which tells us that the affine point $[0 : 4 : 1]$ is indeed a solution. And as we can see, would just as well insert any other representative. For example inserting $(0, 3, 2)$ also satisfies XXX, since

$$3^2 \cdot 2 = 0^3 + 1 \cdot 0 \cdot 2^2 + 1 \cdot 2^3$$

2615 To find the projective representation of E_1 , we first observe that the projective line at infinity
 2616 $[1 : 0 : 0]$ is not a curve point on any projective short Weierstraß curve since it can not satisfy
 2617 XXX for any parameter a and b . So we can exclude it from our consideration.

2618 Moreover a point at infinity $[X : Y : 0]$ can only satisfy equation XXX for any a and b , if
 2619 $X = 0$, which implies that the only point at infinity relevant for short Weierstrass elliptic curves
 2620 is $[0 : 1 : 0]$, since $[0 : k : 0] = [0 : 1 : 0]$ for all k from the finite field. So we can exclude all points
 2621 at infinity except the point $[0 : 1 : 0]$.

So all points that remain are the affine points $[X : Y : 1]$. Inserting all of them into XXX we get the set of all projective curve points as

$$E_1(\mathbb{F}_5\mathbb{P}^2) = \{[0 : 1 : 0], [0 : 1 : 1], [2 : 1 : 1], [3 : 1 : 1], [4 : 2 : 1], [4 : 3 : 1], [0 : 4 : 1], [2 : 4 : 1], [3 : 4 : 1]\}$$

2622 If we compare this with the affine representation we see that there is a 1:1 correspondence
 2623 between the points in the affine representation XXX and the affine points in projective geometry
 2624 and that the point $[0 : 1 : 0]$ represents the additional point \mathcal{O} in the projective representation.

2625 *Exercise 37.* Compute the projective representation of the pen-jubjub curve and the logarithmic
 2626 order of its large prime order subgroup with respect to the generator $(7, 11)$.

Projective Group law As we have seen in XXX, one of the key properties of an elliptic curve is that it comes with a definition of a group law on the set of its rational points, described geometrically by the chord and tangent rule. This rule was kind of intuitive, with the exception of the distinguished point at infinity, which appeared whenever the chord or the tangent did not have a third intersection point with the curve.

One of the key features of projective coordinates is now, that in projective space it is guaranteed that any chord will always intersect the curve in three points and any tangent will intersect in two points including the tangent point. So the geometric picture simplifies as we don't need to consider external symbols and associated cases.

Again, it can be shown that the points of an elliptic curve in projective space form a commutative group with respect to the tangent and chord rule, such that the projective point $[0 : 1 : 0]$ is the neutral element and the additive inverse of a point $[X : Y : Z]$ is given by $[X : -Y : Z]$. The addition law is then usually described by the following algorithm, that minimizes the number of needed additions and multiplications in the base field.

Exercise 38. Compare that affine addition law for short Weierstraß curves with the projective addition rule. Which branch in the projective rule corresponds to which case in the affine law?

Coordinate Transformations As we have seen in example XXX, there was a close relation between the affine and the projective representation of a short Weierstrass curve. This was no accident. In fact from a mathematical point of view projective and affine short Weierstraß curves describe the same thing as there is a one-to-one correspondence (an isomorphism) between both representations for any given parameters a and b .

To specify the isomorphism, let $E(\mathbb{F})$ and $E(\mathbb{FP}^2)$ be an affine and a projective short Weierstraß curve defined for the same parameters a and b . Then the map

$$\Phi : E(\mathbb{F}) \rightarrow E(\mathbb{FP}^2) : \begin{array}{ll} (x, y) & \mapsto [x : y : 1] \\ \mathcal{O} & \mapsto [0 : 1 : 0] \end{array} \quad (5.5)$$

maps points from the affine representation to points from the projective representation of a short Weierstraß curve, that is if the pair of points (x, y) satisfies the affine equation $y^2 = x^3 + ax + b$, then all homogeneous coordinates $(x_1, y_1, z_1) \in [x : y : 1]$ satisfy the projective equation $y_1^2 \cdot z_1 = x_1^3 + ay_1 \cdot z_1^2 + b \cdot z_1^3$. The inverse is given by the map

$$\Phi^{-1} : E(\mathbb{FP}^2) \rightarrow E(\mathbb{F}) : [X : Y : Z] \mapsto \begin{cases} (\frac{X}{Z}, \frac{Y}{Z}) & \text{if } Z \neq 0 \\ \mathcal{O} & \text{if } Z = 0 \end{cases} \quad (5.6)$$

Note the only projective point $[X : Y : Z]$ with $Z \neq 0$ that satisfies XXX is given by the class $[0 : 1 : 0]$.

One key feature of Φ and its inverse is, that it respects the group structure, which means that $\Phi((x_1, y_1) \oplus (x_2, y_2))$ is equal to $\Phi(x_1, y_1) \oplus \Phi(x_2, y_2)$. The same holds true for the inverse map Φ^{-1} .

Maps with these properties are called *group isomorphisms* and from a mathematical point of view the existence of Φ implies, that both definitions are equivalent and implementations can choose freely between both representations.

5.1.2 Montgomery Curves

History and use of them (optimized scalar multiplication)

Algorithm 6 Projective Weierstraß Addition Law

Require: $[X_1 : Y_1 : Z_1], [X_2 : Y_2 : Z_2] \in E(\mathbb{F}_{\mathbb{P}}^2)$

procedure ADD-RULE($[X_1 : Y_1 : Z_1], [X_2 : Y_2 : Z_2]$)

if $[X_1 : Y_1 : Z_1] == [0 : 1 : 0]$ **then**

$[X_3 : Y_3 : Z_3] \leftarrow [X_2 : Y_2 : Z_2]$

else if $[X_2 : Y_2 : Z_2] == [0 : 1 : 0]$ **then**

$[X_3 : Y_3 : Z_3] \leftarrow [X_1 : Y_1 : Z_1]$

else

$U_1 \leftarrow Y_2 \cdot Z_1$

$U_2 \leftarrow Y_1 \cdot Z_2$

$V_1 \leftarrow X_2 \cdot Z_1$

$V_2 \leftarrow X_1 \cdot Z_2$

if $V_1 == V_2$ **then**

if $U_1 \neq U_2$ **then** $[X_3 : Y_3 : Z_3] \leftarrow [0 : 1 : 0]$

else

if $Y_1 == 0$ **then** $[X_3 : Y_3 : Z_3] \leftarrow [0 : 1 : 0]$

else

$W \leftarrow a \cdot Z_1^2 + 3 \cdot X_1^2$

$S \leftarrow Y_1 \cdot Z_1$

$B \leftarrow X_1 \cdot Y_1 \cdot S$

$H \leftarrow W^2 - 8 \cdot B$

$X' \leftarrow 2 \cdot H \cdot S$

$Y' \leftarrow W \cdot (4 \cdot B - H) - 8 \cdot Y_1^2 \cdot S^2$

$Z' \leftarrow 8 \cdot S^3$

$[X_3 : Y_3 : Z_3] \leftarrow [X' : Y' : Z']$

end if

end if

else

$U = U_1 - U_2$

$V = V_1 - V_2$

$W = Z_1 \cdot Z_2$

$A = U^2 \cdot W - V^3 - 2 \cdot V^2 \cdot V_2$

$X' = V \cdot A$

$Y' = U \cdot (V^2 \cdot V_2 - A) - V^3 \cdot U_2$

$Z' = V^3 \cdot W$

$[X_3 : Y_3 : Z_3] \leftarrow [X' : Y' : Z']$

end if

end if

return $[X_3 : Y_3 : Z_3]$

end procedure

Ensure: $[X_3 : Y_3 : Z_3] == [X_1 : Y_1 : Z_1] \oplus [X_2 : Y_2 : Z_2]$

2664 **Affine Montgomery Form** To see what a Montgomery curve in affine coordinates is, let \mathbb{F} be
 2665 a finite field of characteristic > 2 and $A, B \in \mathbb{F}$ two field elements such that $B \neq 0$ and $A^2 \neq 4$.
 2666 Then a **Montgomery elliptic curve** $M(\mathbb{F})$ over \mathbb{F} in its affine representation is the set

$$M(\mathbb{F}) = \{(x, y) \in \mathbb{F} \times \mathbb{F} \mid B \cdot y^2 = x^3 + A \cdot x^2 + x\} \cup \{\mathcal{O}\} \quad (5.7)$$

2667 of all pairs of field elements $(x, y) \in \mathbb{F} \times \mathbb{F}$, that satisfy the Montgomery cubic equation $B \cdot y^2 =$
 2668 $x^3 + A \cdot x^2 + x$, together with a distinguishid symbol \mathcal{O} , called the **point at infinity**.

Despite the fact that Montgomery curves look different then short Weierstrass curve, they are in fact just a special way to describe certain short Weierstrass curves. In fact every curve in affine Montgomery form can be transformed into an elliptic curve in Weierstrass form. To see that assume that a curve in Montgomery form $By^2 = x^3 + Ax^2 + x$ is given. The associated Weierstrass form is then

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

2669 On the other hand, an elliptic curve $E(\mathbb{F})$ over base field \mathbb{F} in Weierstrass form $y^2 = x^3 +$
 2670 $ax + b$ can be converted to Montgomery form if and only if the following conditions hold:

- 2671 • The number of points on $E(\mathbb{F})$ is divisible by 4
- 2672 • The polynomial $z^3 + az + b \in \mathbb{F}[z]$ has at least one root $z_0 \in \mathbb{F}$
- 2673 • $3z_0^2 + a$ is a quadratic residue in \mathbb{F} .

When these conditions are satisfied, then for $s = (\sqrt{3z_0^2 + a})^{-1}$ the equivalent Montgomery curve is defined by the equation

$$sy^2 = x^3 + (3z_0s)x^2 + x$$

2674 If those properties are meet it is therefore possible to transform certain Weierstrass curve into
 2675 Montgomery form. In the following example we will look at our pen-jubjub curve again and
 2676 show that it is actually a Montgomery curve.

2677 *Example 77.* Consider the prime field \mathbb{F}_{13} and the pen-jubjub curve PJJ_13 from example XXX.
 2678 To see that it is a Montgomery curve, we have to check the properties from XXX:

Since the order of PJJ_13 is 20, which is divisible by 4, the first requirement is meet. Next, since $a = 8$ and $b = 8$, we have check if the polynomial $P(z) = z^3 + 8z + 8$ has a root in \mathbb{F}_{13} . We simply evaluate P at all numbers $z \in \mathbb{F}_{13}$ a find that $P(4) = 0$, so a root is given by $z_0 = 4$. In a last step we have to check, that $3 \cdot z_0^2 + a$ has a root in \mathbb{F}_{13} . We compute

$$\begin{aligned} 3z_0^2 + a &= 3 \cdot 4^2 + 8 \\ &= 3 \cdot 3 + 8 \\ &= 9 + 8 \\ &= 4 \end{aligned}$$

To see if 4 is a quadratic residue, we can use Eulers criterium XXX to compute the Legendre symbol of 4. We get:

$$\left(\frac{4}{13} \right) = 4^{\frac{13-1}{2}} = 4^6 = 1$$

2679 so 4 indeed has a root in \mathbb{F}_{13} . In fact computing a root of 4 in \mathbb{F}_{13} is easy, since the integer root
 2680 2 of 4 is also one of its roots in \mathbb{F}_{13} . The other root is given by $13 - 4 = 9$.

Now since all requirements are met, we have shown that PJJ_13 is indeed a Montgomery curve and we can use XXX to compute its associated Montgomery form. We compute

$$\begin{aligned}
 s &= \left(\sqrt{3 \cdot z_0^2 + 8} \right)^{-1} \\
 &= 2^{-1} && \# \text{ Fermat's little theorem} \\
 &= 2^{13-2} && \# 2048 \bmod 13 = 7 \\
 &= 7
 \end{aligned}$$

The defining equation for the Montgomery form of our pen-jubjub curve is then given by the following equation

$$\begin{aligned}
 sy^2 &= x^3 + (3z_0s)x^2 + x && \Rightarrow \\
 7 \cdot y^2 &= x^3 + (3 \cdot 4 \cdot 7)x^2 + x && \Leftrightarrow \\
 7 \cdot y^2 &= x^3 + 6x^2 + x
 \end{aligned}$$

So we get the defining parameters as $B = 7$ and $A = 6$ and we can write the pen-jubjub curve in its affine Montgomery representation as

$$PJJ_13 = \{(x, y) \in \mathbb{F}_{13} \times \mathbb{F}_{13} \mid 7 \cdot y^2 = x^3 + 6x^2 + x\} \cup \{\mathcal{O}\}$$

Now that we have the abstract definition of our pen-jubjub curve in Montgomery form, we can compute the set of points, by inserting all pairs $(x, y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ similar to how we computed the curve points in its Weierstraß representation. We get

$$\begin{aligned}
 PJJ_13 = \{ &\mathcal{O}, (0, 0), (1, 4), (1, 9), (2, 4), (2, 9), (3, 5), (3, 8), (4, 4), (4, 9), \\
 &(5, 1), (5, 12), (7, 1), (7, 12), (8, 1), (8, 12), (9, 2), (9, 11), (10, 3), (10, 10)\}
 \end{aligned}$$

2681

```

2682 sage: F13 = GF(13)                                     314
2683 sage: L_MPJJ = []                                       315
2684 .....: for x in F13:                                    316
2685 .....:     for y in F13:                                317
2686 .....:         if F13(7)*y^2 == x^3 + F13(6)*x^2 + x:  318
2687 .....:             L_MPJJ.append((x, y))                319
2688 sage: MPJJ = Set(L_MPJJ)                                320
2689 sage: # does not compute the point at infinity          321

```

2690 **Affine Montgomery coordinate transformation** Comparing the Montgomery representa-
 2691 tion of the previous example with the Weierstraß representation of the same curve, we see that
 2692 there is a 1:1 correspondence between the curve points in both examples. This is no accident. In
 2693 fact if $M_{A,B}$ is a Montgomery curve and $E_{a,b}$ a Weierstraß curve with $a = \frac{3-A^2}{3B^2}$ and $b = \frac{2A^2-9A}{27B^3}$
 2694 then the function

$$\Phi : M_{A,B} \rightarrow E_{a,b} : (x, y) \mapsto \left(\frac{3x+A}{3B}, \frac{y}{B} \right) \quad (5.8)$$

2695 maps all points in Montgomery representation onto the points in Weierstraß representation. This
 2696 map is a 1:1 correspondence (an isomorphism) and its inverse map is given by

$$\Phi^{-1} : E_{a,b} \rightarrow M_{A,B} : (x, y) \mapsto (s \cdot (x - z_0), s \cdot y) \quad (5.9)$$

where z_0 is a root of the polynomial $z^3 + az + b \in \mathbb{F}[z]$ and $s = (\sqrt{3z_0^2 + a})^{-1}$. Using this map, it is therefore possible for implementations of Montgomery curves to freely transit between the Weierstraß and the Montgomery representation. Note however that according to XXX not every Weierstraß curve is a Montgomery curve, as all of the properties from XXX have to be satisfied. The map Φ^{-1} therefore does not always exist.

Example 78. Consider our pen-jubjub curve again. In example XXX we derive its Weierstraß representation and in example XXX we derive its Montgomery representation.

To see how the coordinate transformation Φ works in this example, let's map points from the Montgomery representation onto points from the Weierstraß representation. Inserting for example the point $(0,0)$ from the Montgomery representation XXX into Φ gives

$$\begin{aligned}\Phi(0,0) &= \left(\frac{3 \cdot 0 + A}{3B}, \frac{0}{B} \right) \\ &= \left(\frac{3 \cdot 0 + 6}{3 \cdot 7}, \frac{0}{7} \right) \\ &= \left(\frac{6}{8}, 0 \right) \\ &= (4,0)\end{aligned}$$

So the Montgomery point $(0,0)$ maps to the self inverse point $(4,0)$ of the Weierstraß representation. On the other hand we can use our computations of $s = 7$ and $z_0 = 4$ from XXX, to compute the inverse map Φ^{-1} , which maps point on the Weierstraß representation to points on the Montgomery form. Inserting for example $(4,0)$ we get

$$\begin{aligned}\Phi^{-1}(4,0) &= (s \cdot (4 - z_0), s \cdot 0) \\ &= (7 \cdot (4 - 4), 0) \\ &= (0,0)\end{aligned}$$

So as expected, the inverse map maps the Weierstraß point back to where it came from on the Montgomery form. We can invoke sage to proof that our computation of Φ is correct:

```

2706 sage: # Compute PHI of Montgomery form: 322
2707 sage: L_PHI_MPJJ = [] 323
2708 sage: for (x,y) in L_MPJJ: # LMJJ as defined previously 324
2709     v = (F13(3)*x + F13(6)) / (F13(3)*F13(7)) 325
2710     w = y/F13(7) 326
2711     L_PHI_MPJJ.append((v,w)) 327
2712 sage: PHI_MPJJ = Set(L_PHI_MPJJ) 328
2713 sage: # Computation Weierstrass form 329
2714 sage: C_WPJJ = EllipticCurve(F13,[8,8]) 330
2715 sage: L_WPJJ = [P.xy() for P in C_WPJJ.points() if P.order() > 331
2716     1]
2717 sage: WPJJ = Set(L_WPJJ) 332
2718 sage: # check PHI(Montgomery) == Weierstrass 333
2719 sage: WPJJ == PHI_MPJJ 334
2720 True 335
2721 sage: # check the inverse map PHI^(-1) 336

```



```

2722 sage: L_PHIINV_WPJJ = []
2723 sage: for (v,w) in L_WPJJ:
2724     ....:     x = F13(7) * (v-F13(4))
2725     ....:     y = F13(7) * w
2726     ....:     L_PHIINV_WPJJ.append((x,y))
2727 sage: PHIINV_WPJJ = Set(L_PHIINV_WPJJ)
2728 sage: MPJJ == PHIINV_WPJJ
2729 True

```

Montgomery group law So we see that Montgomery curves are special cases of short Weierstrass curves. As such they have a group structure defined on the set of their points, which can also be derived from a chord and tangent rule. In accordance with short Weierstrass curves, it can be shown that the identity $x_1 = x_2$ implies $y_2 = \pm y_1$, which shows that the following rules are a complete description of the affine addition law.

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

1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P + Q = P$.
2. (Adding inverse elements) If $P = (x,y)$ and $Q = (x, -y)$, the sum is defined as $P + Q = \mathcal{O}$.
3. (Adding non self-inverse equal points) If $P = (x,y)$ and $Q = (x,y)$ with $y \neq 0$, the sum $2P = (x', y')$ is defined by

$$x' = \left(\frac{3x_1^2 + 2Ax_1 + 1}{2By_1} \right)^2 \cdot B - (x_1 + x_2) - A, \quad y' = \frac{3x_1^2 + 2Ax_1 + 1}{2By_1} (x_1 - x') - y_1$$

4. (Adding non inverse different points) If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $x_1 \neq x_2$, the sum $R = P + Q$ with $R = (x_3, y_3)$ is defined by

$$x' = \left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 B - (x_1 + x_2) - A, \quad y' = \frac{y_2 - y_1}{x_2 - x_1} (x_1 - x') - y_1$$

5.1.3 Twisted Edwards Curves

As we have seen in XXX both Weierstrass and Montgomery curves have somewhat complicated addition and doubling laws as many cases have to be distinguished. Those cases translate to branches in computer programs.

In the context of SNARK development two computational models for bounded computations, called *circuits* and *rank-1 constraint systems*, are used and program-branches are undesirably costly, when implemented in those models. It is therefore advantageous to look for curves with an addition/doubling rule, that requires no branches and as few field operations as possible.

Twisted Edwards curves are particularly useful here as a subclass of these curves has a compact and easy to implement addition law that works for all points, including the point at infinity. Implementing that rule therefore needs no branching.

Twisted Edwards Form To see what an affine **twisted Edwards curve** looks like, let \mathbb{F} be a finite field of characteristic > 2 and $a, d \in \mathbb{F} \setminus \{0\}$ two non zero field elements with $a \neq d$. Then a **twisted Edwards elliptic curve** in its affine representation is the set

$$E(\mathbb{F}) = \{(x, y) \in \mathbb{F} \times \mathbb{F} \mid a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2\} \quad (5.10)$$

of all pairs (x, y) from $\mathbb{F} \times \mathbb{F}$, that satisfy the twisted Edwards equation $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$. A twisted Edwards curve is called an Edwards curve (non twisted), if the parameter a is equal to 1 and is called a **snark friendly** twisted Edwards curve if the parameter a is a quadratic residue and the parameter d is a quadratic non residue.

As we can see from the definition, affine twisted Edwards curve look somewhat different from Weierstraß curves as their affine representation does not need a special symbol to represent the point at infinity. In fact we will see that the pair $(0, 1)$ is always a point on any twisted Edwards curve and that it takes the role of the point at infinity.

Despite the different looks however, twisted Edwards curves are equivalent to Montgomery curves in the sense that for every twisted Edwards curve there is a Montgomery curve and a way to map the points of one curve in a 1:1 correspondence onto the other and vice versa. To see that assume that a curve in twisted Edwards form $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$ is given. The associated Montgomery curve is then defined by the Montgomery equation

$$\frac{4}{a-d} y^2 = x^3 + \frac{2(a+d)}{a-d} \cdot x^2 + x \quad (5.11)$$

On the other hand a Montgomery curve $By^2 = x^3 + Ax^2 + x$ with $B \neq 0$ and $A^2 \neq 4$ can give rise to a twisted Edwards curve defined by the equation

$$\left(\frac{A+2}{B}\right)x^2 + y^2 = 1 + \left(\frac{A-2}{B}\right)x^2 y^2 \quad (5.12)$$

Recalling from XXX that Montgomery curves are just a special class of Weierstraß, we now know that twisted Edwards curve are special Weierstraß curves too. So the more general way to describe elliptic curves are Weierstraß curves.

Example 79. Consider the pen jubjub curve from example XXX again. We know from XXX that it is a Montgomery curve and since Montgomery curves are equivalent to twisted Edwards curve, we want to write that curve in twisted Edwards form. We use XXX and compute the parameters a and d as

$$\begin{aligned} a &= \frac{A+2}{B} & \# \text{ insert } A=6 \text{ and } B=7 \\ &= \frac{8}{7} = 3 & \# 7^{-1} = 2 \\ d &= \frac{A-2}{B} \\ &= \frac{4}{7} = 8 \end{aligned}$$

So we get the defining parameters as $a = 3$ and $d = 8$. Since our goal is to use this curve later on in implementations of pen and paper snarks, let's show that tiny-jubjub is moreover a *snark friendly* twisted Edwards curve. To see that, we have to show that a is a quadratic residue and

d is a quadratic non residue. We therefore compute the Legendre symbols of a and d using the Euler criterium. We get

$$\left(\frac{3}{13}\right) = 3^{\frac{13-1}{2}} = 3^6 = 1$$

$$\left(\frac{8}{13}\right) = 8^{\frac{13-1}{2}} = 8^6 = 12 = -1$$

which proofs that tiny-jubjub is snark friendly. We can write the tiny-jubjub curve in its affine twisted Edwards representation as

$$TJJ_{13} = \{(x, y) \in \mathbb{F}_{13} \times \mathbb{F}_{13} \mid 3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2\}$$

Now that we have the abstract definition of our pen-jubjub curve in twisted Edwards form, we can compute the set of points, by inserting all pairs $(x, y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ similar to how we computed the curve points in its Weierstraß or Edwards representation. We get

$$PJJ_{13} = \{(0, 1), (0, 12), (1, 2), (1, 11), (2, 6), (2, 7), (3, 0), (5, 5), (5, 8), (6, 4), (6, 9), (7, 4), (7, 9), (8, 5), (8, 8), (10, 0), (11, 6), (11, 7), (12, 2), (12, 11)\}$$

2775

```

2776 sage: F13 = GF(13)                                     345
2777 sage: L_EPJJ = []                                       346
2778 ..... for x in F13:                                     347
2779 .....     for y in F13:                                   348
2780 .....         if F13(3)*x^2 + y^2 == 1+ F13(8)*x^2*y^2: 349
2781 .....             L_EPJJ.append((x, y))                 350
2782 sage: EPJJ = Set(L_EPJJ)                                 351

```

Twisted Edwards group law As we have seen, twisted Edwards curves are equivalent to Montgomery curves and as such also have a group law. However, in contrast to Montgomery and Weierstraß curves, the group law of snark friendly twisted Edwards curves can be described by single computation, that works in all cases, no matter if we add the neutral element, inverse, or if have to double a point. To see how the group law looks like, first observe that the point $(0, 1)$ is a solution to $a \cdot x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2$ for any curve. The sum of any two points $(x_1, y_1), (x_2, y_2)$ on an Edwards curve $E(\mathbb{F})$ is then given by

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

2783 and it can be shown that the point $(0, 1)$ serves as the neutral element and the inverse of a point
 2784 (x_1, y_1) is given by $(-x_1, y_1)$.

Example 80. Lets look at the tiny-jubjub curve in Edwards form from example XXX again. As we have seen, this curve is given by

$$PJJ_{13} = \{(0, 1), (0, 12), (1, 2), (1, 11), (2, 6), (2, 7), (3, 0), (5, 5), (5, 8), (6, 4), (6, 9), (7, 4), (7, 9), (8, 5), (8, 8), (10, 0), (11, 6), (11, 7), (12, 2), (12, 11)\}$$

To get an understanding of the twisted Edwards addition law, let's first add the neutral element $(0, 1)$ to itself. We apply the group law XXX and get

$$\begin{aligned}(0, 1) \oplus (0, 1) &= \left(\frac{0 \cdot 1 + 1 \cdot 0}{1 + 8 \cdot 0 \cdot 0 \cdot 1 \cdot 1}, \frac{1 \cdot 1 - 3 \cdot 0 \cdot 0}{1 - 8 \cdot 0 \cdot 0 \cdot 1 \cdot 1} \right) \\ &= (0, 1)\end{aligned}$$

So as expected, adding the neutral element to itself gives the neutral element again. Now let's add the neutral element to some other curve point. We get

$$\begin{aligned}(0, 1) \oplus (8, 5) &= \left(\frac{0 \cdot 5 + 1 \cdot 8}{1 + 8 \cdot 0 \cdot 8 \cdot 1 \cdot 5}, \frac{1 \cdot 5 - 3 \cdot 0 \cdot 8}{1 - 8 \cdot 0 \cdot 8 \cdot 1 \cdot 5} \right) \\ &= (8, 5)\end{aligned}$$

Again as expected adding the neutral element to any element will give the element again. Given any curve point (x, y) , we know that the inverse is given by $(-x, y)$. To see how the addition of a point to its inverse works out we therefore compute

$$\begin{aligned}(5, 5) \oplus (8, 5) &= \left(\frac{5 \cdot 5 + 5 \cdot 8}{1 + 8 \cdot 5 \cdot 8 \cdot 5 \cdot 5}, \frac{5 \cdot 5 - 3 \cdot 5 \cdot 8}{1 - 8 \cdot 5 \cdot 8 \cdot 5 \cdot 5} \right) \\ &= \left(\frac{12 + 1}{1 + 5}, \frac{12 - 3}{1 - 5} \right) \\ &= \left(\frac{0}{6}, \frac{12 + 10}{1 + 8} \right) \\ &= \left(0, \frac{9}{9} \right) \\ &= (0, 1)\end{aligned}$$

So adding a curve point to its inverse gives the neutral element, as expected. As we have seen from these examples the twisted Edwards addition law handles edge cases particularly nice and in a unified way.

5.2 Elliptic Curves Pairings

As we have seen in XXX some groups come with the notation of a so called pairing map, which is a non-degenerate bilinear map, from two groups into another group.

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

We start out by defining elliptic curve pairings and discussing a simple application which bears some resemblance to 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.

Elliptic curves have a few structures, like the Weil or the Tate map, that qualifies as pairing.

Embedding Degrees As we will see in what follows, every elliptic curve gives rise to a pairing map. However as we will see in example XXX, not every such pairing is efficiently computable. So in order to distinguish curves with efficiently computable pairings from the rest, we need to start with an introduction to the so called **embedding degree** of a curve.

To understand this term, let \mathbb{F} be a finite field, $E(\mathbb{F})$ an elliptic curve over \mathbb{F} , and n a prime number that divides the order of $E(\mathbb{F})$. The embedding degree of $E(\mathbb{F})$ with respect to n is then the smallest integer k such that n divides $q^k - 1$.

Fermat's little theorem XXX implies, that every curve has at least *some* embedding degree k , since at least $k = n - 1$ is always a solution to the congruency $q^k \equiv 1 \pmod{n}$ which implies that the remainder of the integer division of $q^k - 1$ by n is 0.

Example 81. To get a better intuition of the embedding degree, lets consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We know from XXX that the order of $E_1(\mathbb{F}_5)$ is 9 and since the only prime factor of 9 is 3, we compute the ebedding degree of $E_1(\mathbb{F}_5)$ with respect to 3.

To find that embedding degree we have to find the smallest integer k , such that 3 divides $q^k - 1 = 5^k - 1$. We try and increment until we find a proper k .

$$\begin{array}{ll} k = 1: 5^1 - 1 = 4 & \text{not divisible by 3} \\ k = 2: 5^2 - 1 = 24 & \text{divisible by 3} \end{array}$$

So we know that the embedding degree of $E_1(\mathbb{F}_5)$ is 2 relative to the the prime factor 3.

Example 82. Lets consider the tiny jubjub curve *TJJ_13* from example XXX. We know from XXX that the order of *TJJ_13* is 20 and that the order therefore has two prime factors. A "large" prime factor 5 and a small prime factor 2.

We start by computing the ebedding degree of *TJJ_13* with respect to the large prime factor 5. To find that embedding degree we have to find the smallest integer k , such that 5 divides $q^k - 1 = 13^k - 1$. We try and increment until we find a proper k .

$$\begin{array}{ll} k = 1: 13^1 - 1 = 12 & \text{not divisible by 5} \\ k = 2: 13^2 - 1 = 168 & \text{not divisible by 5} \\ k = 3: 13^3 - 1 = 2196 & \text{not divisible by 5} \\ k = 4: 13^4 - 1 = 28560 & \text{divisible by 5} \end{array}$$

So we know that the embedding degree of *TJJ_13* is 4 relative to the the prime factor 5.

In real world applications, like on pairing friendly elliptic curves as for example BLS_12-381, usually only the embedding degree of the large prime factor are relevant, which in case of out tiny-jubjub curve, is represented by 5. It should however be noted that every prime factor of a curves order has its own notation of embedding degree despite the fact that this is mostly irrelevant in applications.

To find the embedding degree of the small prime factor 2 we have to find the smallest integer k , such that 2 divides $q^k - 1 = 13^k - 1$. We try and increment until we find a proper k .

$$k = 1: 13^1 - 1 = 12 \quad \text{divisible by 2}$$

So we know that the embedding degree of *TJJ_13* is 1 relative to the the prime factor 2. So as we have seen, different prime factors can have different embedding degrees in general.

```

sage: p = 13                                     352
sage: # large prime factor                       353
sage: n = 5                                       354
sage: for k in range(1,5): # Fermat's little theorem 355
.....:     if (p^k-1)%n == 0:                   356
.....:         break                             357

```

```

2830 sage: k                                     358
2831 4                                           359
2832 sage: # small prime factor                 360
2833 sage: n = 2                               361
2834 sage: for k in range(1,2): # Fermat's little theorem 362
2835 .....:     if (p^k-1)%n == 0:             363
2836 .....:         break                       364
2837 sage: k                                     365
2838 1                                           366

```

2839 *Example 83.* To give an example of a cryptographically secure real world elliptic curve that does
 2840 not have a small embedding degree lets look at curve secp256k1 again. We know from XXX
 2841 that the order of this curve is a prime number, so we only have a single embedding degree.

2842 To test potential embedding degrees k , say in the range $1 \dots 1000$, we can invoke sage and
 2843 compute:

```

2844 sage: p = 1157920892373161954235709850086879078532699846656405 367
2845 64039457584007908834671663
2846 sage: n = 1157920892373161954235709850086879078528375642790749 368
2847 04382605163141518161494337
2848 sage: for k in range(1,1000):                 369
2849 .....:     if (p^k-1)%n == 0:                 370
2850 .....:         break                           371
2851 sage: k                                       372
2852 999                                         373

```

2853 So we see that secp256k1 has at least no embedding degree $k < 1000$, which renders secp256k1
 2854 as a curve that has no small embedding degree. A property that is of importance later on.

2855 **Elliptic Curves over extension fields** Suppose that p is a prime number and \mathbb{F}_p its associated
 2856 prime field. We know from XXX, that the fields \mathbb{F}_{p^m} are extensions of \mathbb{F}_p in the sense that \mathbb{F}_p
 2857 is a subfield of \mathbb{F}_{p^m} . This implies that we can extend the affine plane an elliptic curve is defined
 2858 on, by changing the base field to any extension field. To be more precise let $E(\mathbb{F}) = \{(x, y) \in$
 2859 $\mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + a \cdot x + b\}$ be an affine short Weierstrass curve, with parameters a and b taken
 2860 from \mathbb{F} . If \mathbb{F}' is any extension field of \mathbb{F} , then we extend the domain of the curve by defining

$$E(\mathbb{F}') = \{(x, y) \in \mathbb{F}' \times \mathbb{F}' \mid y^2 = x^3 + a \cdot x + b\} \quad (5.13)$$

2861 So while we did not change the defining parameters, we consider curve points from the affine
 2862 plane over the extension field now. Since $\mathbb{F} \subset \mathbb{F}'$ it can be shown that the original elliptic curve
 2863 $E(\mathbb{F})$ is a sub curve of the extension curve $E(\mathbb{F}')$.

Example 84. Consider the prime field \mathbb{F}_5 from example XXX and the elliptic curve $E_1(\mathbb{F}_5)$ from
 example XXX. Since we know from XXX that \mathbb{F}_{5^2} is an extension field of \mathbb{F}_5 , we can extend
 the definition of $E_1(\mathbb{F}_5)$ to define a curve over \mathbb{F}_{5^2} :

$$E_1(\mathbb{F}_{5^2}) = \{(x, y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + x + 1\}$$

2864 Since \mathbb{F}_{5^2} contains 25 points, in order to compute the set $E_1(\mathbb{F}_{5^2})$, we have to try $25 \cdot 25 = 625$
 2865 pairs, which is probably a bit to much for the avarage motivated reader. Instaed we involve sage
 2866 to compute the curve for us. To do so choose the representation of \mathbb{F}_{5^2} from XXX. We get:

```

2867 sage: F5= GF(5) 374
2868 sage: F5t.<t> = F5[] 375
2869 sage: P = F5t(t^2+2) 376
2870 sage: P.is_irreducible() 377
2871 True 378
2872 sage: F5_2.<t> = GF(5^2, name='t', modulus=P) 379
2873 sage: E1F5_2 = EllipticCurve(F5_2, [1,1]) 380
2874 sage: E1F5_2.order() 381
2875 27 382

```

So curve $E_1(\mathbb{F}_{5^2})$ consist of 27 points, in contrast to curve $E_1(\mathbb{F}_5)$, which consists of 9 points. Printing the points gives

$$\begin{aligned}
 E_1(\mathbb{F}_{5^2}) = \{ & \mathcal{O}, (0,4), (0,1), (3,4), (3,1), (4,3), (4,2), (2,4), (2,1), \\
 & (4t+3, 3t+4), (4t+3, 2t+1), (3t+2, t), (3t+2, 4t), \\
 & (2t+2, t), (2t+2, 4t), (2t+1, 4t+4), (2t+1, t+1), \\
 & (2t+3, 3), (2t+3, 2), (t+3, 2t+4), (t+3, 3t+1), \\
 & (3t+1, t+4), (3t+1, 4t+1), (3t+3, 3), (3t+3, 2), (1, 4t) \}
 \end{aligned}$$

2876 As we can see, curve $E_1(\mathbb{F}_5)$ sits inside curve $E(\mathbb{F}_{5^2})$, which is implied from \mathbb{F}_5 being a subfield
2877 of \mathbb{F}_{5^2} .

2878 **Full Torsion groups** The fundamental theorem of finite cyclic groups XXX implies, that
2879 every prime factor n of a cyclic groups order defines a subgroup of the size of the prime factor.
2880 We called such a subgroup an n -torsion group. We have seen many of those subgroups in the
2881 examples XXX and XXX.

2882 Now when we consider elliptic curve extensions as defined in XXX, we could ask, what
2883 happens to the n -torsion groups in the extension. One might intuitively think that their expan-
2884 sion just parallels the extension of the curve. For example when $E(\mathbb{F}_p)$ is a curve over prime
2885 field \mathbb{F}_p , with some n -torsion group \mathbb{G} and when we extend the curve to $E(\mathbb{F}_{p^m})$, then there is a
2886 bigger n -torsion group, such that \mathbb{G} is a subgroup. Naively this would make sense, as $E(\mathbb{F}_p)$ is
2887 a subcurve of $E(\mathbb{F}_{p^m})$.

2888 However the real situation is a bit more surprising then that. To see that, let \mathbb{F}_p be a prime
2889 field and $E(\mathbb{F}_p)$ an elliptic curve of order r , with embedding degree k and n -torsion group
2890 $E(\mathbb{F}_p)[n]$ for same prime factor n of r . Then it can be shown that the n -torsion group $E(\mathbb{F}_{p^m})[n]$
2891 of a curve extension is equal to $E(\mathbb{F}_p)[n]$, as long as the power m is less then the embedding
2892 degree k of $E(\mathbb{F}_p)$.

2893 However for the prime power p^m , for any $m \geq k$, $E(\mathbb{F}_{p^m})[n]$ is strictly larger then $E(\mathbb{F}_p)[n]$
2894 and contains $E(\mathbb{F}_p)[n]$ as a subgroup. We call the n -torsion group $E(\mathbb{F}_{p^k})[n]$ of the extension of
2895 E over \mathbb{F}_{p^k} the **full n -torsion group** of that elliptic curve. It can be shown that it contains n^2
2896 many elements and consists of $n+1$ subgroups, one of which is $E(\mathbb{F}_p)[n]$.

2897 So roughly speaking, when we consider towers of curve extensions $E(\mathbb{F}_{p^m})$, ordered by the
2898 prime power m , then the n -torsion group stays constant for every level m small then the embed-
2899 ding degee, while it suddenly blossoms into a larger group on level k , with $n+1$ subgroups and
2900 it then stays like that for any level m larger then k . In other words, once the extension field is
2901 big enough to find on emore point of order n (that is not defined over the base field), then we
2902 actually find all of the points in the full torsion group.

2903 *Example 85.* Consider curve $E_1(\mathbb{F}_5)$ again. We know that it contains a 3-torsion group and that
 2904 the embedding degree of 3 is 2. From this we can deduce that we can find the full 3-torsion
 2905 group $E_1[3]$ in the curve extension $E_1(\mathbb{F}_{5^2})$, the latter of which we computed in XXX.

2906 Since that curve is small, in order to find the full 3-torsion, we can loop through all elements
 2907 of $E_1(\mathbb{F}_{5^2})$ and check check the defining equation $[3]P = \mathcal{O}$. Invoking sage we compute

```
2908 sage: INF = E1F5_2(0) # Point at infinity 383
2909 sage: L_E1_3 = [] 384
2910 sage: for p in E1F5_2: 385
2911 ....:     if 3*p == INF: 386
2912 ....:         L_E1_3.append(p) 387
2913 sage: E1_3 = Set(L_E1_3) # Full 3-torsion set 388
```

we get

$$E_1[3] = \{\mathcal{O}, (1, t), (1, 4t), (2, 1), (2, 4), (2t+1, t+1), (2t+1, 4t+4), (3t+1, t+4), (3t+1, 4t+1)\}$$

2914 *Example 86.* Consider the tiny jubjub curve from example XXX. we know from XXX that it
 2915 contains a 5-torsion group and that the embedding degree of 5 is 4. This implies that we can
 2916 find the full 5-torsion group $TJJ_13[5]$ in the curve extension $TJJ_13(\mathbb{F}_{13^4})$.

2917 To compute the full torsion, first observe that since \mathbb{F}_{13^4} contains 28561 element, computing
 2918 $TJJ_13(\mathbb{F}_{13^4})$ means checking $28561^2 = 815730721$ elements. From each of these curve points
 2919 P , we then have to check the equation $[5]P = \mathcal{O}$. Doing this for 815730721 is a bit to slow even
 2920 on a computer.

2921 Fortunate sage has a way to loop through points of given order efficiently. The following
 2922 sage code then gives a way to compute the full torsion group:

```
2923 sage: # define the extension field 389
2924 sage: F13= GF(13) # prime field 390
2925 sage: F13t.<t> = F13[] # polynomials over t 391
2926 sage: P = F13t(t^4+2) # irreducible polynomial of degree 4 392
2927 sage: P.is_irreducible() 393
2928 True 394
2929 sage: F13_4.<t> = GF(13^4, name='t', modulus=P) # F_{13^4} 395
2930 sage: TJJF13_4 = EllipticCurve(F13_4, [8, 8]) # tiny jubjub 396
2931 extension
2932 sage: # compute the full 5-torsion 397
2933 sage: L_TJJF13_4_5 = [] 398
2934 sage: INF = TJJF13_4(0) 399
2935 sage: for P in INF.division_points(5): # [5]P == INF 400
2936 ....:     L_TJJF13_4_5.append(P) 401
2937 sage: len(L_TJJF13_4_5) 402
2938 25 403
2939 sage: TJJF13_4_5 = Set(L_TJJF13_4_5) 404
```

2940 So as expected we get a group that contains $5^2 = 25$ elements. As its rather tedious to write this
 2941 group down and as we don't need in what follows we skipp writing it. To see that the embedding
 2942 degree 4 is actually the smallerst prime power to find the full 5-torsion group, lets compute the
 2943 5-torsion group over of the tiny-jubjub curve the extension field \mathbb{F}_{13^3} . We get

```
2944 sage: # define the extension field 405
```



```

2945 sage: P = F13t(t^3+2) # irreducible polynomial of degree 3 406
2946 sage: P.is_irreducible() 407
2947 True 408
2948 sage: F13_3.<t> = GF(13^3, name='t', modulus=P) # F_{13^3} 409
2949 sage: TJJF13_3 = EllipticCurve(F13_3,[8,8]) # tiny jubjub 410
2950 extension
2951 sage: # compute the 5-torsion 411
2952 sage: L_TJJF13_3_5 = [] 412
2953 sage: INF = TJJF13_3(0) 413
2954 sage: for P in INF.division_points(5): # [5]P == INF 414
2955 .....:     L_TJJF13_3_5.append(P) 415
2956 sage: len(L_TJJF13_3_5) 416
2957 5 417
2958 sage: TJJF13_3_5 = Set(L_TJJF13_3_5) # full $$$-torsion 418

```

2959 So as we can see the 5-torsion group of tiny-jubjub over \mathbb{F}_{13^3} is equal to the 5-torsion group of
2960 tiny-jubjub over \mathbb{F}_{13} itself.

2961 *Example 87.* Lets look at curve Secp256k1. We know from XXX that the curve is of some
2962 prime order r and hence the only n -torsion group to consider is the curve itself. So the curve
2963 group is the r -torsion.

However in order to find the full r -torsion of Secp256k1, we need to compute the embedding
degree k and as we have seen in XXX it is at least not small. We know from Fermat's little
theorem that a finite embedding degree must exist, though. It can be shown that it is given by

$$k = 192986815395526992372618308347813175472927379845817397100860523586360249056$$

2964 which is a 256bit number. So the embedding degree is huge, which implies that the field ex-
2965 tension \mathbb{F}_{p^k} is huge too. To understand how big \mathbb{F}_{p^k} is, recall that an element of \mathbb{F}_{p^m} can be
2966 represented as a string $[x_0, \dots, x_m]$ of m elements, each containing a number from the prime
2967 field \mathbb{F}_p . Now in the case of Secp256k1, such a representation has k -many entries, each of 256
2968 bits in size. So without any optimizations, representing such an element would need $k \cdot 256$ bits,
2969 which is too much to be represented in the observable universe.

2970 **Torsion-Subgroups** As we have stated above, any full n -torsion group contains $n + 1$ cyclic
2971 subgroups, two of which are of particular interest in pairing based elliptic curve cryptography.
2972 To characterize these groups we need to consider the so called *Frobenious* endomorphism

$$\pi : E(\mathbb{F}) \rightarrow E(\mathbb{F}) : \begin{array}{ccc} (x, y) & \mapsto & (x^p, y^p) \\ \mathcal{O} & \mapsto & \mathcal{O} \end{array} \quad (5.14)$$

2973 of an elliptic curve $E(\mathbb{F})$ over some finite field \mathbb{F} of characteristic p . It can be shown that π maps
2974 curve points to curve points. The first thing to note is that in case that \mathbb{F} is a prime field, the
2975 Frobenious endomorphism acts trivially, since $(x^p, y^p) = (x, y)$ on prime fields, due to Fermat's
2976 little theorem XX. So the Frobenious map is more interesting over prime field extensions.

2977 With the Frobenious map at hand, we can now characterise two important subgroups of the
2978 full n -torsion. The first subgroup is the n -torsion group that already exists in the curve over the
2979 base field. In pairing based cryptography this group is usually written as \mathbb{G}_1 , assuming that the
2980 prime factor ' n ' in the definition is implicitly given. Since we know that the Frobenious map,
2981 acts trivially on curve over the prime field we can define \mathbb{G}_1 as:

$$\mathbb{G}_1[n] := \{(x, y) \in E[n] \mid \pi(x, y) = (x, y)\} \quad (5.15)$$

In more mathematical terms this definition means, that \mathbb{G}_1 is the *Eigenspace* of the Frobenious map with respect to the *Eigenvalue* 1.

Now it can be shown, that there is another subgroup of the full n -torsion group that can be characterized by the Frobenious map. In the context of so called type 3 pairing based cryptography this subgroup is usually called \mathbb{G}_2 and it defined as

$$\mathbb{G}_2[n] := \{(x, y) \in E[n] \mid \pi(x, y) = [p](x, y)\} \quad (5.16)$$

So in mathematical terms \mathbb{G}_2 is the *Eigenspace* of the Frobenious map with respect to the *Eigenvalue* p .

Notation and Symbols 9. If the prime factor n of the curves order is clear from the context, we sometimes simply write \mathbb{G}_1 and \mathbb{G}_2 to mean $\mathbb{G}_1[n]$ and $\mathbb{G}_2[n]$, respectively.

It should be noted however that sometimes other definitions of \mathbb{G}_2 appear in the literature, however in the context of pairing based cryptography, this is the most common one. It is particularly useful, as we can define hash functions that map into \mathbb{G}_2 , which is not possible for all subgroups of the full n -torsion.

Example 88. Consider the curve $E_1(\mathbb{F}_5)$ from example XXX again. As we have seen this curve has embedding degree $k = 2$ and a full 3-torsion group is given by

$$E_1[3] = \{\mathcal{O}, (2, 1), (2, 4), (1, t), (1, 4t), (2t + 1, t + 1), (2t + 1, 4t + 4), (3t + 1, t + 4), (3t + 1, 4t + 1)\}$$

According to the general theory, $E_1[3]$ contains 4 subgroups and we can characterise the subgroups \mathbb{G}_1 and \mathbb{G}_2 using the Frobenious endomorphism. Unfortunately at the time of this writing sage did have a predefined Frobenious endomorphism for elliptic curves, so we have to use the Frobenious endomorphism of the underlying field as a temporary workaround. We compute

```
sage: L_G1 = []
sage: for P in E1_3:
.....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
.....:     if P == PiP:
.....:         L_G1.append(P)
sage: G1 = Set(L_G1)
```

So as expected the group $\mathbb{G}_1 = \{\mathcal{O}, (2, 4), (2, 1)\}$ is identical to the 3-torsion group of the (un-extended) curve over the prime field $E_1(\mathbb{F}_5)$. We can use almost the same algorithm to compute the group \mathbb{G}_2 and get

```
sage: L_G2 = []
sage: for P in E1_3:
.....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
.....:     pP = 5*P # [5]P
.....:     if pP == PiP:
.....:         L_G2.append(P)
sage: G2 = Set(L_G2)
```

so we compute the the second subgroup of the full 3-torsion group of curve E_1 as the set $\mathbb{G}_2 = \{\mathcal{O}, (1, t), (1, 4t)\}$.

Example 89. Considering the tiny-jubjub curve *TJJ_13* from example XXX. In example XXX we computed its full 5 torsion, which is a group that has 6 subgroups. We compute $G1$ using sage as:

```

3021 sage: L_TJJ_G1 = [] 432
3022 sage: for P in TJJF13_4_5: 433
3023     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P) 434
3024     if P == PiP: 435
3025         L_TJJ_G1.append(P) 436
3026 sage: TJJ_G1 = Set(L_TJJ_G1) 437

```

3027 We get $\mathbb{G}_1 = \{\mathcal{O}, (7, 2), (8, 8), (8, 5), (7, 11)\}$

```

3028 sage: L_TJJ_G1 = [] 438
3029 sage: for P in TJJF13_4_5: 439
3030     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P) 440
3031     pP = 13*P # [5]P 441
3032     if pP == PiP: 442
3033         L_TJJ_G1.append(P) 443
3034 sage: TJJ_G1 = Set(L_TJJ_G1) 444

```

3035 $\mathbb{G}_2 = \{\mathcal{O}, (9t^2 + 7, t^3 + 11t), (9t^2 + 7, 12t^3 + 2t), (4t^2 + 7, 5t^3 + 10t), (4t^2 + 7, 8t^3 + 3t)\}$

3036 *Example 90.* Consider Bitcoin's curve Secp256k1 again. Since the group \mathbb{G}_1 is identical to the
3037 torsion group of the unextended curve and since Secp256k1 has prime order, we know, that in
3038 thi case \mathbb{G}_1 is identical to Secp256k1. It is however infeasible not just to compute \mathbb{G}_2 itself, but
3039 to even compute an avarage element of \mathbb{G}_2 as elements need to much storage to be representable
3040 in this universe.

3041 **The Weil Pairing** In this part we consider a pairing function defined on the subgroups $\mathbb{G}_1[r]$
3042 and $\mathbb{G}_2[r]$ of the full r -torsion $E[r]$ of a short Weierstraß elliptic curve. To be more precise let
3043 $E(\mathbb{F}_p)$ be an elliptic curve of embedding degree k , such that r is a prime factor of its order. Then
3044 the **Weil pairing** is a bilinear, non-degenerate map

$$e(\cdot, \cdot) : \mathbb{G}_1[r] \times \mathbb{G}_2[r] \rightarrow \mathbb{F}_{p^k} ; (P, Q) \mapsto (-1)^r \cdot \frac{f_{r,P}(Q)}{f_{r,Q}(P)} \quad (5.17)$$

3045 where the extension field elements $f_{r,P}(Q), f_{r,Q}(P) \in \mathbb{F}_{p^k}$ are computed by **Miller's algorithm**:
3046 Understanding in detail how the algorithm works requires the concept of *divisors*, which we
3047 don't really need in this book. The interested reader might look at [REFERENCES]

3048 In real world application of pairing friendly elliptic curves, the embedding degree is usuall
3049 a small number like 2, 4, 6 or 12 and the number r is the largest prime factor of the curves order.

3050 *Example 91.* Consider curve $E_1(\mathbb{F}_5)$ from example XXX. Since the only prime factor of the
3051 groups order is 3 we can not compute the Weil pairing on this group using our definition of
3052 Miller's algorithm. In fact since \mathbb{G}_1 is of order 3, executing the if statement on line XXX will
3053 lead to a division by zero error in the computation of the slope m .

Example 92. Consider the tiny-jubjub curve $TJJ_{13}(\mathbb{F}_{13})$ from example XXX again. We want
to instantiate the general definition of the Weil pairing for this example. To do so, recall that
we have see in example XXX, its embedding degree is 4 and that we have the following type-3
pairing groups:

$$\begin{aligned} \mathbb{G}_1 &= \{\mathcal{O}, (7, 2), (8, 8), (8, 5), (7, 11)\} \\ \mathbb{G}_2 &= \{\mathcal{O}, (9t^2 + 7, t^3 + 11t), (9t^2 + 7, 12t^3 + 2t), (4t^2 + 7, 5t^3 + 10t), (4t^2 + 7, 8t^3 + 3t)\} \end{aligned}$$

Algorithm 7 Miller's algorithm for short Weierstraß curves $y^2 = x^3 + ax + b$

Require: $r > 3$, $P \in E[r]$, $Q \in E[r]$ and

$b_0, \dots, b_t \in \{0, 1\}$ with $r = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_t \cdot 2^t$ and $b_t = 1$

procedure MILLER'S ALGORITHM(P, Q)

if $P = \mathcal{O}$ or $Q = \mathcal{O}$ or $P = Q$ **then**

return $f_{r,P}(Q) \leftarrow (-1)^r$

end if

$(x_T, y_T) \leftarrow (x_P, y_P)$

$f_1 \leftarrow 1$

$f_2 \leftarrow 1$

for $j \leftarrow t - 1, \dots, 0$ **do**

$m \leftarrow \frac{3 \cdot x_T^2 + a}{2 \cdot y_T}$

$f_1 \leftarrow f_1^2 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$

$f_2 \leftarrow f_2^2 \cdot (x_Q + 2x_T - m^2)$

$x_{2T} \leftarrow m^2 - 2x_T$

$y_{2T} \leftarrow -y_T - m \cdot (x_{2T} - x_T)$

$(x_T, y_T) \leftarrow (x_{2T}, y_{2T})$

if $b_j = 1$ **then**

$m \leftarrow \frac{y_T - y_P}{x_T - x_P}$

$f_1 \leftarrow f_1 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$

$f_2 \leftarrow f_2 \cdot (x_Q + (x_P + x_T) - m^2)$

$x_{T+P} \leftarrow m^2 - x_T - x_P$

$y_{T+P} \leftarrow -y_T - m \cdot (x_{T+P} - x_T)$

$(x_T, y_T) \leftarrow (x_{T+P}, y_{T+P})$

end if

end for

$f_1 \leftarrow f_1 \cdot (x_Q - x_T)$

return $f_{r,P}(Q) \leftarrow \frac{f_1}{f_2}$

end procedure

3054 where \mathbb{G}_1 and \mathbb{G}_2 are subgroups of the full 5-torsion found in the curve $TJJ_13(\mathbb{F}_{13^4})$. The
 3055 type-3 Weil pairing is a map $e(\cdot, \cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{F}_{13^4}$. From the first if-statement in Miller's
 3056 algorithm, we can deduce that $e(\mathcal{O}, Q) = 1$ as well as $e(P, \mathcal{O}) = 1$ for all arguments $P \in \mathbb{G}_1$ and
 3057 $Q \in \mathbb{G}_2$. So in order to compute a non-trivial Weil pairing we choose the arguments $P = (7, 2)$
 3058 and $Q = (9t^2 + 7, 12t^3 + 2t)$.

3059 In order to compute the pairing $e((7, 2), (9t^2 + 7, 12t^3 + 2t))$ we have to compute the exten-
 3060 sion field elements $f_{5,P}(Q)$ and $f_{5,Q}(P)$ applying Miller's algorithm. Do do so first observe that
 3061 we have $5 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$, so we get $t = 2$ as well as $b_0 = 1$, $b_1 = 0$ and $b_2 = 1$. The
 3062 loop therefore needs to be executed two times.

Computing $f_{5,P}(Q)$, we initiate $(x_T, y_T) = (7, 2)$ as well as $f_1 = 1$ and $f_2 = 1$. Then

j	b_j	m	f_1	f_2	x_{2T}	y_{2T}	x_{T+P}	y_{T+P}
1	·							

$$\begin{aligned}
 m &= \frac{3 \cdot x_T^2 + a}{2 \cdot y_T} \\
 &= \frac{3 \cdot 2^2 + 1}{2 \cdot 4} = \frac{3}{3} \\
 &= 1
 \end{aligned}$$

$$\begin{aligned}
 f_1 &= f_1^2 \cdot (y_Q - y_T - m \cdot (x_Q - x_T)) \\
 &= 1^2 \cdot (t - 4 - 1 \cdot (1 - 2)) = t - 4 + 1 \\
 &= t + 2
 \end{aligned}$$

$$\begin{aligned}
 f_2 &= f_2^2 \cdot (x_Q + 2x_T - m^2) \\
 &= 1^2 \cdot (1 + 2 \cdot 2 - 1^2) = (1 + 4 - 1) \\
 &= 4
 \end{aligned}$$

$$\begin{aligned}
 x_{2T} &= m^2 - 2x_T \\
 &= 1^2 - 2 \cdot 2 = -3 \\
 &= 2
 \end{aligned}$$

$$\begin{aligned}
 y_{2T} &= -y_T - m \cdot (x_{2T} - x_T) \\
 &= -4 - 1 \cdot (2 - 2) = -4 \\
 &= 1
 \end{aligned}$$

So we update $(x_T, y_T) = (2, 1)$ and since $b_0 = 1$ we have to execute the if statement on line XXX in the for loop. However since we only loop a single time, we don't need to compute

y_{T+P} , since we only need the updated x_T in the final step. We get:

$$\begin{aligned} m &= \frac{y_T - y_P}{x_T - x_P} \\ &= \frac{1 - 4}{2 - x_P} \end{aligned}$$

$$f_1 = f_1 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$$

$$f_2 = f_2 \cdot (x_Q + (x_P + x_T) - m^2)$$

$$x_{T+P} = m^2 - x_T - x_P$$

5.3 Hashing to Curves

Elliptic curve cryptography frequently requires the ability to hash data onto elliptic curves. If the order of the curve is not a prime number hashing to prime number subgroups is also of importance. In the context of pairing friendly curves it is also sometimes necessary to hash specifically onto the group \mathbb{G}_1 or \mathbb{G}_2 .

As we have seen in XXX, many general methods are known to hash into groups in general and finite cyclic groups in particular. As elliptic groups are cyclic those methods can be utilized in this case, too. However in what follows we want to describe some methods special to elliptic curves, that are frequently used in applications.

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 modular arithmetic 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, that is a try and increment hash as described in XXX is used. If the first *try* of hashing to the field does not result in a valid curve point, the counter

Algorithm 8 Hash-to- $E(\mathbb{F}_r)$ **Require:** $r \in \mathbb{Z}$ with $r.\text{nbits}() = k$ and $s \in \{0, 1\}^*$ **Require:** Curve equation $y^2 = x^3 + ax + b$ over \mathbb{F}_r **procedure** TRY-AND-INCREMENT(r, k, s) $c \leftarrow 0$ **repeat** $s' \leftarrow s || c.\text{bits}()$ $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \dots + H(s')_k \cdot 2^k$ $x \leftarrow z^3 + a \cdot z + b$ $c \leftarrow c + 1$ **until** $z < r$ and $x^{\frac{r-1}{2}} \bmod r = 1$ **if** $H(s')_{k+1} == 0$ **then** $y \leftarrow \sqrt{x} \#(\text{root in } \mathbb{F}_r)$ **else** $y \leftarrow r - \sqrt{x} \#(\text{root in } \mathbb{F}_r)$ **end if****return** (x, y) **end procedure****Ensure:** $(x, y) \in E(\mathbb{F}_r)$

is *incremented* and the hashing is repeated again. This is done until a valid curve point is found eventually.

This method has the advantage that it 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 will fail to generate a quadratic residue. Fortunately it is possible to make the probability for this arbitrarily small by choosing large enough counters and relying on the (approximate) uniformity of the hash-to-field function.

If the curve is not of prime order, the result will be a general curve point that might not be in the "large" prime order subgroup. A so called *cofactor clearing* step is then necessary to project the curve point onto the subgroup. This is done by scalar multiplication with the cofactor of prime order with respect to the curves order.

Example 93. Consider the tiny jubjub curve from example XXX. We want to construct a try and increment hash function, that hashes a binary string s of arbitrary length onto the large prime order subgroup of size 5.

Since the curve as well as our target subgroup are defined over the field \mathbb{F}_{13} and the binary representation of 13 is $13.\text{bits}() = 1101$, we apply SHA256 from sage's crypto library on the concatenation $s || c$ for some binary counter string and use the first 4 bits of the image to try to hash into \mathbb{F}_{13} . In case we are able to hash to a value z , such that $z^3 + 8 \cdot z + 8$ is a quadratic residue in \mathbb{F}_{13} , we use the 5-th bit to decide which of the two possible roots of $z^3 + 8 \cdot z + 8$ we will choose as the y -coordinate. The result is then a curve point different from the point at infinity. To project it to a point of \mathbb{G}_1 , we multiply it with the cofactor 4. If the result is still not the point at infinity, it is the result of the hash.

To make this concrete let $s = '10011001111010110100000111'$ be our binary string that we want to hash onto \mathbb{G}_1 . We use a 4-bit binary counter, starting at zero, i.e we choose $c = 0000$. Invoking sage we define the try-hash function as

```
sage: import hashlib
```

```

3119 sage: def try_hash(s,c): 446
3120 .....:     s_1 = s+c 447
3121 .....:     hasher = hashlib.sha256(s_1.encode('utf-8')) 448
3122 .....:     digest = hasher.hexdigest() 449
3123 .....:     d = Integer(digest,base=16) 450
3124 .....:     sign = d.str(2)[-5:-4] 451
3125 .....:     d = d.str(2)[-4:] 452
3126 .....:     z = Integer(d,base=2) 453
3127 .....:     return (z,sign) 454
3128 sage: try_hash('10011001111010110100000111','0000') 455
3129 (15, '1') 456

```

3130 As we can see, our first attempt to hash into \mathbb{F}_{13} was not successful as 15 is not a number in
3131 \mathbb{F}_{13} , so we increment the binary counter by 1 and try again:

```

3132 sage: try_hash('10011001111010110100000111','0001') 457
3133 (3, '0') 458

```

3134 And we find a hash into \mathbb{F}_{13} . However this point is not guaranteed to define a curve point. To see
3135 that we insert $z = 3$ into the right side of the Weierstraß equation of the tiny.jubjub curve and
3136 compute $3^3 + 8 \cdot 3 + 8 = 7$, but 7 is not a quadratic residue in \mathbb{F}_{13} since $7^{\frac{13-1}{2}} = 7^6 = 12 = -1$.
3137 So 3 is not a suitable point and we have to increment the counter two more times:

```

3138 sage: try_hash('10011001111010110100000111','0010') 459
3139 (3, '0') 460
3140 sage: try_hash('10011001111010110100000111','0011') 461
3141 (6, '1') 462

```

Since $6^3 + 8 \cdot 6 + 8 = 12$ and we have $\sqrt{12} \in \{5, 8\}$, we finally found the valid x coordinate
 $x = 6$ for the curve point hash. Now since the sign bit of this hash is 1, we choose the larger
root $y = 8$ as the y -coordinate and get the hash

$$H('10011001111010110100000111') = (6, 8)$$

which is a valid curve point on the tiny jubjub curve. Now in order to project it onto the
"large" prime order subgroup we have to do cofactor clearing, that is we have to multiply the
point with the cofactor 4. We get

$$[4](6, 8) = \mathcal{O}$$

3142 so the hash value is still not right. We therefore have to increment the counter two times again,
3143 until we finally find a correct hash to \mathbb{G}_1

```

3144 sage: try_hash('10011001111010110100000111','0100') 463
3145 (0, '1') 464
3146 sage: try_hash('10011001111010110100000111','0101') 465
3147 (12, '0') 466

```

Since $12^3 + 8 \cdot 12 + 8 = 12$ and we have $\sqrt{12} \in \{5, 8\}$, we found another valid x coordinate
 $x = 12$ for the curve point hash. Now since the sign bit of this hash is 0, we choose the smaller
root $y = 5$ as the y -coordinate and get the hash

$$H('10011001111010110100000111') = (12, 5)$$

which is a valid curve point on the tiny jubjub curve and in order to project it onto the "large" prime order subgroup we have to do cofactor clearing, that is we have to multiply the point with the cofactor 4. We get

$$[4](12, 5) = (8, 5)$$

3148 So hashing the binary string '10011001111010110100000111' onto \mathbb{G}_1 gives the hash value
3149 (8, 5) as a result.

3150 5.4 Constructing elliptic curves

3151 Cryptographically secure elliptic curves like Secp256k1 from example XXX are known for
3152 quite some time. In the latest advancements of cryptography, it is however often necessary to
3153 design and instantiate elliptic curves from scratch, that satisfy certain very specific properties.

3154 For example, in the context of SNARK development it was necessary to design a curve that
3155 can be efficiently implemented inside of a so called circuit, in order to enable primitives like
3156 elliptic curve signature schemes in a zero knowledge proof. Such a curve is give by the Baby-
3157 JubJub curve [XXX] and we have paralled its definition by introducing the tiny-JubJub curve
3158 from example XX. As we have seen those curves are instances of so called twisted Edwards
3159 curves and as such have easy to implement addition laws that work without branching. However
3160 we introduced the tiny-jubjub curve out of thin air, as we just gave the curve parameters without
3161 explaining how we came up with them.

3162 Another requirement in the context of many so called pairing based zero knowledge proofing
3163 systems is the existing of a suitable, pairing friendly curve with a specified security level and
3164 a low embedding degree as defined in XXX. Famous examples are the BLS_12 or the NMT
3165 curves.

3166 The major goal of this section is to explain the most important method to design elliptic
3167 curves with predefined properties from sratch, called the *complex multiplication method*. We
3168 will apply this method in section to synthezise a particular BLS_6 curve, the most insecure
3169 BLS_6 curve, which will serve as the main curve to build our pen and paper snarks on. As we
3170 will see, this curve has a "large" prime factor subgroup of order 13, which implies, that we can
3171 use our tiny-jubjub curve to implement certain elliptic curve cryptographic primitives in circuits
3172 over that BLS_6 curve.

3173 Before we introduce the complex multiplication method, we have to explain a few properties
3174 of elliptic curves that are of key importants in understanding the complex multiplication method.

3175 **The Trace of Frobenious** To understand the complex multiplication method of elliptic curve,
3176 we have to define the so called *trace* of an elliptic curve first.

3177 We know from XXX that elliptic curves over finite fields are cyclic groups of finite order.
3178 An interesting question therefore is, if it is possible to estimate the number of elements that
3179 curve contains. Since an affine short Weierstraß curve consists of pairs (x, y) of elements from a
3180 finite field \mathbb{F}_q plus the point at infinity and the field \mathbb{F}_q contains q elements, the number of curve
3181 points can not be arbitrarily large, since it can contain at most $q^2 + 1$ many elements.

3182 There is however a more precise estimation, usually called the **Hasse bound**. To understand
3183 it, let $E(\mathbb{F}_q)$ be an affine short Weierstraß curve over a finite field \mathbb{F}_w of order q and let $|E(\mathbb{F}_q)|$
3184 be the order of the curve. Then there is an integer $t \in \mathbb{Z}$ called the **trace of Frobenious** of the
3185 curve, such that $|t| \leq 2\sqrt{q}$ and

$$|E(\mathbb{F})| = q + 1 - t \tag{5.18}$$

3186 A positive trace therefore implies, that the curve contains less points then the underlying field
 3187 and a negative trace means that the curve contains more point. However the estimation $|t| \leq 2\sqrt{q}$
 3188 implies that the difference is not very large in either direction and the number of elements in an
 3189 elliptic curve is always approximately in the same order of magnitude as the size of the curve's
 3190 basefield.

Example 94. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We know that it contains 9 curve points. Since the order of \mathbb{F}_5 is 5 we compute the trace of $E_1(\mathbb{F})$ to be $t = -3$, since the Hasse bound is given by

$$9 = 5 + 1 - (-3)$$

3191 And indeed we have $|t| \leq 2\sqrt{q}$, since $\sqrt{5} > 2.23$ and $|-3| = 3 \leq 4.46 = 2 \cdot 2.23 < 2 \cdot \sqrt{5}$.

Example 95. To compute the trace of the tiny-jubjub curve, oberse from example XXX, that the order of PJJ_13 is 20. Since the order of \mathbb{F}_{13} is 13, we can therefore use the Hasse bound and compute the trace as $t = -6$, since

$$20 = 13 + 1 - (-6)$$

3192 Again we have $|t| \leq 2\sqrt{q}$, since $\sqrt{13} > 3.60$ and $|-6| = 6 \leq 7.20 = 2 \cdot 3.60 < 2 \cdot \sqrt{13}$.

Example 96. To compute the trace of Secp256k1, recall from example XXX, that this curve is defined over a prime field with p elements and that the order of that group is given by r , with

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

Using the Hesse bound $r = p + 1 - t$, we therefore compute $t = p + 1 - r$, which gives the trace of curve Secp256k1 as

$$t = 432420386565659656852420866390673177327$$

3193 So as we can see Secp256k1 contains less elements then its underlying field. However the
 3194 difference is tiny, since the order of Secp256k1 is in the same order of magnitude as the order of
 3195 the underlying field. Compared to p and r , t is tiny.

```

3196 sage: p = 1157920892373161954235709850086879078532699846656405 467
3197       64039457584007908834671663
3198 sage: r = 1157920892373161954235709850086879078528375642790749 468
3199       04382605163141518161494337
3200 sage: t = p + 1 - r 469
3201 sage: t.nbits() 470
3202 129 471
3203 sage: abs(RR(t)) <= 2*sqrt(RR(p)) 472
3204 True 473

```

3205 **The j -invariant** As we have seen in XXX two elliptic curve $E_1(\mathbb{F})$ defined by $y^2 = x^3 + ax + b$
 3206 and $E_2(\mathbb{F})$ defined by $y^2 + a'x + b'$ are strictly isomorphic, if and only if there is a quadratic
 3207 residue $d \in \mathbb{F}$, such that $a' = ad^2$ and $b' = bd^3$.

3208 There is however a more general way to classify elliptic curves over finite fields \mathbb{F}_q , based
 3209 on the so called j -invariant of an elliptic curve:

$$j(E(\mathbb{F}_q)) = (1728 \bmod q) \frac{4 \cdot a^3}{4 \cdot a^3 + (27 \bmod q) \cdot b^2} \quad (5.19)$$

3210 with $j(E(\mathbb{F}_q)) \in \mathbb{F}_q$. We will not go into the details of the j -invariant, but state only, that two
 3211 elliptic curves $E_1(\mathbb{F})$ and $E_2(\mathbb{F}')$ are isomorphic over the algebraic closures of \mathbb{F} and \mathbb{F}' , if and
 3212 only if $\overline{\mathbb{F}} = \overline{\mathbb{F}'}$ and $j(E_1) = j(E_2)$.

3213 So the j -invariant is an important tool to classify elliptic curves and it is needed in the com-
 3214 plex multiplication method to decide on an actual curve instantiation, that implements ab-
 3215 stractly chosen properties.

Example 97. Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We compute its j -invariant as

$$\begin{aligned} j(E_1(\mathbb{F}_5)) &= (1728 \bmod 5) \frac{4 \cdot 1^3}{4 \cdot 1^3 + (27 \bmod 5) \cdot 1^2} \\ &= 3 \frac{4}{4+2} \\ &= 3 \cdot 4 = 2 \end{aligned}$$

Example 98. Consider the elliptic curve PJJ_13 from example XXX. We compute its j -invariant as

$$\begin{aligned} j(E_1(\mathbb{F}_5)) &= (1728 \bmod 13) \frac{4 \cdot 8^3}{4 \cdot 8^3 + (27 \bmod 13) \cdot 8^2} \\ &= 12 \cdot \frac{4 \cdot 5}{4 \cdot 5 + 1 \cdot 12} \\ &= 12 \cdot \frac{7}{7+12} \\ &= 12 \cdot 7 \cdot 6^{-1} \\ &= 12 \cdot 7 \cdot 11 \\ &01 \end{aligned}$$

3216 *Example 99.* Consider Secp256k1 from example XXX. We compute its j -invariant using sage:

3217

```
3218 sage: p = 1157920892373161954235709850086879078532699846656405 474
3219       64039457584007908834671663
3220 sage: F = GF(p) 475
3221 sage: j = F(1728) * ( (F(4) * F(0)^3) / (F(4) * F(0)^3 + F(27) * F(7)^2) ) 476
3222 sage: j == F(0) 477
3223 True 478
```

3224 **The Complex Multiplication Method** As we have seen in the previous sections, elliptic
 3225 curves have various defining properties, like their order and their prime factors, the embedding
 3226 degree, or the cardinality of the base field. The so called *complex multiplication* (CM) gives a
 3227 practical method for constructing elliptic curves with pre-defined restrictions on the order and
 3228 the base field.

3229 The method usually starts by choosing a base field \mathbb{F}_q of the curve $E(\mathbb{F}_q)$ we want to con-
 3230 struct, such that $q = p^m$ for some prime number p and counting number $m \in \mathbb{N}$ with $m \geq 1$. We
 3231 assume $p > 3$ to simplify things in what follows.

3232 Next the trace of Frobenius $t \in \mathbb{Z}$ of the curve is chosen, such that p and t are coprime,
 3233 i.e. such that $\gcd(p, t) = 0$ holds true. The choice of t also defines the curves order r , since

$r = p + 1 - t$ by the Hasse bound XXX, so choosing t , will define the large order subgroup as well as all small cofactors. r has to be defined in such a way, that the elliptic curve meets the security requirements of the application it is designed for.

Note that the choice of p and t also determines the embedding degree k of any prime order subgroup of the curve, since k is defined as the smallest number, such that the prime order n divides the number $q^k - 1$.

In order for the complex multiplication method to work, both q and t can not be arbitrary, but must be choosen in such a way that two additional integers $D \in \mathbb{Z}$ and $v \in \mathbb{Z}$ exist, such that $D < 0$ as well as $D \bmod 4 = 0$ or $D \bmod 4 = 1$ and the equation

$$4q = t^2 + |D|v^2 \quad (5.20)$$

holds. If those numbers exist, we call D the *CM-discriminant* and we know that we can construct a curve $E(\mathbb{F}_q)$ over a finite field \mathbb{F}_q , such that the order of the curve is $|E(\mathbb{F}_q)| = q + 1 - t$.

It is the content of the complex multiplication method to actually construct such a curve, that is finding the parameters a and b from \mathbb{F}_q in the defining Weierstraß equation, such that the curve has the desired order r .

Finding solutions to equation XXX, can be achieved in different ways, which we will not look much into. In general it can be said, that there are well known constrections for elliptic curve families like the BLS (ECT) families, that provides families of solutions. In what follows we will look at one type curves the BLS-family, which gives an entire rane of solutions.

Assuming that proper parameters q , t , D and v are found, we have to compute the so called *Hilbert class polynomial* $H_D \in \mathbb{Z}[x]$ of the CM-discriminant D , which is a polynomial with integer coefficients. To do so, we first have to compute the following set:

$$ICG(D) = \{(A, B, C) \mid A, B, C \in \mathbb{Z}, D = B^2 - 4AC, \gcd(A, B, C) = 1,$$

$$|B| \leq A \leq \sqrt{\frac{|D|}{3}}, A \leq C, \text{ if } B < 0 \text{ then } |B| < A < C\}$$

One way to compute this set, is to first compute the integer $A_{max} = \text{Floor}(\sqrt{\frac{|D|}{3}})$, then loop through all the integers A in the range $[0, \dots, A_{max}]$ as well as through all the integers B in the range $[-A_{max}, \dots, A_{max}]$ and to see if there is an integer C , that satisfies $D = B^2 - 4AC$ and the rest of the requirements in XXX.

To compute the Hilbert class polynomial, the so called *j-function* (or *j-invariant*) is needed, which is a complex function defined on the upper half \mathbb{H} of the complex plane \mathbb{C} , usually written as

$$j : \mathbb{H} \rightarrow \mathbb{C} \quad (5.21)$$

Roughly speaking what this means is that the *j-functions* takes complex numbers $(x + i \cdot y)$ with positive imaginary part $y > 0$ as inputs and returns a complex number $j(x + i \cdot y)$ as result.

For the sake of this book it is not important to actually understand the *j-function* and we can use sage to compute it in a similar way as we would use sage to computer any other well known function. It should be noted however, that the computation of the *j-function* in sage is sometimes prone to precision errors. For example the *j-function* has a root in $\frac{-1+i\sqrt{3}}{2}$, which sage only approximates. Therefore using sage to compute the *j-function*, we need to take precision loss into account and eventually round to the nearest integer.

```
sage: z = ComplexField(100)(0,1)
```

```
sage: z # (0+1i)
```

479

480

3269	1.00000000000000000000000000000000*I	481
3270	sage: elliptic_j(z)	482
3271	1728.0000000000000000000000000000	483
3272	sage: # j-function only defined for positive imaginary	484
3273	arguments	
3274	sage: z = ComplexField(100)(1,-1)	485
3275	sage: try:	486
3276: elliptic_j(z)	487
3277: except PariError:	488
3278: pass	489
3279	sage: # root at (-1+i sqrt(3))/2	490
3280	sage: z = ComplexField(100)(-1,sqrt(3))/2	491
3281	sage: elliptic_j(z)	492
3282	-2.6445453750358706361219364880e-88	493
3283	sage: elliptic_j(z).imag().round()	494
3284	0	495
3285	sage: elliptic_j(z).real().round()	496
3286	0	497

3287 With a way to compute the j -function and the precomputed set $ICG(D)$ at hand, we can now
3288 compute the Hilbert class polynomial as

$$H_D(x) = \Pi_{(A,B,C) \in ICG(D)} \left(x - j \left(\frac{-B + \sqrt{D}}{2A} \right) \right) \quad (5.22)$$

So what we do is we loop over all elements (A, B, C) from the set $ICG(D)$ and compute the j -function at the point $\frac{-B+\sqrt{D}}{2A}$, where D is the CM-discriminant that we decided in a previous step. The result then defines a factor of the Hilbert class polynomial and all factors are multiplied together.

It can be shown, that the Hilbert class polynomial is an integer polynomial, but actual computations need high precision arithmetics to avoid approximation errors, that usually occur in computer approximations of the j -function as shown above. So in case the calculated Hilbert class polynomial does not have integer coefficients, we need to round the result to the nearest integer. Given that the precision we used was high enough, the result will be correct.

In the next step we use the Hilbert class polynomial $H_D \in \mathbb{Z}[x]$ and project it to a polynomial $H_{D,q} \in \mathbb{F}_q[x]$ with coefficients in the base field \mathbb{F}_q as chosen in the first step. We do this by simply computing the new coefficients as the old coefficients modulus p , that is if $H_D(x) = a_mx^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0$ we compute the q -modulus of each coefficient $\tilde{a}_j = a_j \bmod p$ which defines the *projected Hilbert class polynomial* as

$$H_{D,p}(x) = \tilde{a}_m x^m + \tilde{a}_{m-1} x^{m-1} + \dots + \tilde{a}_1 x + \tilde{a}_0$$

We then search for roots of $H_{D,p}$, since every root j_0 of $H_{D,p}$ defines a family of elliptic curves over \mathbb{F}_q , which all have a j -invariant XXX equal to j_0 . We can pick any root and all of them will lead to proper curves eventually.

3301 However some of the curves with the correct j -invariant might have an order different from
3302 the one we initially decided on. So we need a way to decide on a curve with the correct order.

To compute such a curve, we have to distinguish a few different cases, based on our choice of the root j_0 and of the CM-discriminant D . If $j_0 \neq 0$ or $j_0 \not\equiv 1728 \pmod{q}$ we compute

3305 $c_1 = \frac{j_0}{(1728 \bmod q) - j_0}$ and then we chose some arbitrary quadratic non-residue $c_2 \in \mathbb{F}_q$ and some
 3306 arbitrary cubic non residue $c_3 \in \mathbb{F}_q$.

3307 The following table is guranteed to define a curve with the correct order $r = q + 1 - t$, for
 3308 the trace of Frobenious t we initally decided on:

- 3309 • Case $j_0 \neq 0$ and $j_0 \neq 1728 \bmod q$. A curve with the correct order is defined by one of the
 3310 following equations

$$y^2 = x^3 + 3c_1x + 2c_1 \quad \text{or} \quad y^2 = x^3 + 3c_1c_2^2x + 2c_1c_2^3 \quad (5.23)$$

- 3311 • Case $j_0 = 0$ and $D \neq -3$. A curve with the correct order is defined by one of the following
 3312 equations

$$y^2 = x^3 + 1 \quad \text{or} \quad y^2 = x^3 + c_2^3 \quad (5.24)$$

- Case $j_0 = 0$ and $D = -3$. A curve with the correct order is defined by one of the following
 equations

$$\begin{aligned} y^2 &= x^3 + 1 \quad \text{or} \quad y^2 = x^3 + c_2^3 \quad \text{or} \\ y^2 &= x^3 + c_3^2 \quad \text{or} \quad y^2 = c_3^2c_2^3 \quad \text{or} \\ y^2 &= x^3 + c_3^{-2} \quad \text{or} \quad y^2 = x^3 + c_3^{-2}c_2^3 \end{aligned}$$

- 3313 • Case $j_0 = 1728 \bmod q$ and $D \neq -4$. A curve with the correct order is defined by one of
 3314 the following equations

$$y^2 = x^3 + x \quad \text{or} \quad y^2 = x^3 + c_2^2x \quad (5.25)$$

- Case $j_0 = 1728 \bmod q$ and $D = -4$. A curve with the correct order is defined by one of
 the following equations

$$\begin{aligned} y^2 &= x^3 + x \quad \text{or} \quad y^2 = x^3 + c_2x \quad \text{or} \\ y^2 &= x^3 + c_2^2x \quad \text{or} \quad y^2 = x^3 + c_2^3x \end{aligned}$$

3315 To decide the proper defining Weierstraß equation, we therefore have to compute the order of
 3316 any of the potential curves above and then choose the one that fits out initial requirements.
 3317 Since it can be shown that the Hilbert class polynomials for the CM-discriminants $D = -3$ and
 3318 $D = -4$ are given by $H_{-3,q}(x) = x$ and $H_{-4,q} = x - (1728 \bmod q)$ (EXERCISE) the previous
 3319 cases are exhaustive.

3320 To summarize, using the complex multiplication method, it is possible to synthesize elliptic
 3321 curve with predefined order over predefined base fields from scratch. However the curves that
 3322 are constructed this way are just some representatives of a larger class of curves, all of which
 3323 have the same order. In applications it is therefore sometimes more advantageous to choose a
 3324 different representative from that class. To do so recall from XXX, that any curve defined by
 3325 the Weierstraß equation $y^2 = x^3 + axb$ is isomorphic to a curve of the form $y^2 = x^3 + ad^2x + bd^3$
 3326 for some quadratic residue $d \in \mathbb{F}_q$.

3327 So in order to find a nice representative (e.g. with small parameters a and b) in a last step,
 3328 the designer might choose a quadratic residue d such that the transformed curve looks the way
 3329 they wanted it.

3330 *Example 100.* Consider curve $E_1(\mathbb{F}_5)$ from example XXX. We want to use the complex multi-
 3331 plication method to derive that curve from scratch. Since $E_1(\mathbb{F}_5)$ is a curve of order $r = 9$ over
 3332 the prime field of order $q = 5$, we know from example XX that its trace of Frobenius is $t = -3$,
 3333 which also implies that q and $|t|$ are coprime.

We then have to find parameters $D, v \in \mathbb{Z}$ with $D < 0$ and $D \bmod 4 = 0$ or $D \bmod 4 = 1$, such that $4q = t^2 + |D|v^2$ holds. We get

$$\begin{aligned} 4q &= t^2 + |D|v^2 && \Rightarrow \\ 20 &= (-3)^2 + |D|v^2 && \Leftrightarrow \\ 11 &= |D|v^2 \end{aligned}$$

3334 Now since 11 is a prime number, the only solution is $|D| = 11$ and $v = 1$ here. So $D = -11$ and
 3335 since the Euklidean division of -11 by 4 is $-11 = -3 \cdot 4 + 1$ we have $-11 \bmod 4 = 1$, which
 3336 shows that $D = -11$ is a proper choice.

3337 In the next step, we have to compute the Hilbert class polynomial H_{-11} and to do so, we
 3338 first have to find the set $ICG(D)$. To compute that set, observe, that since $\sqrt{\frac{|D|}{3}} \approx 1.915 < 2$, we
 3339 know from $A \leq \sqrt{\frac{|D|}{3}}$ and $A \in \mathbb{Z}$ that A must be either 0 or 1.

3340 For $A = 0$, we know $B = 0$ from the constraint $|B| \leq A$, but in this case there can be no C
 3341 satisfying $-11 = B^2 - 4AC$. So we try $A = 1$ and deduce $B \in \{-1, 0, 1\}$ from the constraint
 3342 $|B| \leq A$. The case $B = -1$ can be excluded since then $B < 0$ has to imply $|B| < A$. In addition,
 3343 the case $B = 0$ can be exclude as there can be integer C with $-11 = -4C$ since 11 is a prime
 3344 number.

This leaves the case $B = 1$ and we compute $C = 3$ from the equation $-11 = 1^2 - 4C$, which gives the solution $(A, B, C) = (1, 1, 3)$ and we get

$$ICG(D) = \{(1, 1, 3)\}$$

With the set $ICG(D)$ at hand we can compute the Hilbert class polynomial of $D = -11$. To do so, we have to insert the term $\frac{-1 + \sqrt{-11}}{2 \cdot 1}$ into the j -function. To do so first observe that $\sqrt{-11} = i\sqrt{11}$, where i is the imaginary unit, defined by $i^2 = -1$. Using this, we can invoke `sagemath` to compute the j -invariant and get

$$H_{-11}(x) = x - j\left(\frac{-1 + i\sqrt{11}}{2}\right) = x + 32768$$

So as we can see, in this particular case, the Hilbert class polynomial is a linear function with a single integer corefficient. In the next step we have to project it onto a polynomial from $\mathbb{F}_5[x]$, by computing the modular 5 remainder of the corefficients 1 and 32768. We get $32768 \bmod 5 = 3$ from which follows that the projected Hilbert class polynomial is

$$H_{-11,5}(x) = x + 3$$

considered as a polynomial from $\mathbb{F}_5[x]$. As we can see the only root of this polynomial is $j = 2$, since $H_{-11,5}(2) = 2 + 3 = 0$. We therefore have a situation with $j \neq 0$ and $j \neq 1728$, which tells us that we have to compute the parameter

$$c_1 = \frac{2}{1728 - 2}$$

in modular 5 arithmetics. Since $1728 \bmod 5 = 3$, we get $c_1 = 2$. Then we have to check if the curve $E(\mathbb{F}_5)$ defined by the Weierstraß $y^2 = x^3 + 3 \cdot 2x + 2 \cdot 2$ has the correct order. We invoke sage and find that the order is indeed 9, so it is a curve with the required parameters and we are done.

Note however that in real world applications, it might be usefull to choose parameters a and b that have certain properties, e.g. to be as small as possible. As we know from XXX, choosing any quadratic residue $d \in \mathbb{F}_5$ gives a curve of the same order defined by $y^2 = x^3 + ak^2x + bk^3$. Since 4 is a quadratic residue in \mathbb{F}_4 , we can transform the curve defined by $y^2 = x^3 + x + 4$ into the curve $y^2 = x^3 + 4^2 + 4 \cdot 4^3$ which gives

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

which is the curve $E_1(\mathbb{F}_5)$, that we used extensively throughout this book. So using the complex multiplication method, we were able to derive a curve with specific properties from scratch.

Example 101. Consider the tiny jubjub curve TJJ_13 from example XXX. We want to use the complex multiplication method to derive that curve from scratch. Since TJJ_13 is a curve of order $r = 20$ over the prime field of order $q = 13$, we know from example XX that its trace of Frobenius is $t = -6$, which also implies that q and $|t|$ are coprime.

We then have to find parameters $D, v \in \mathbb{Z}$ with $D < 0$ and $D \bmod 4 = 0$ or $D \bmod 4 = 1$, such that $4q = t^2 + |D|v^2$ holds. We get

$$\begin{aligned} 4q &= t^2 + |D|v^2 && \Rightarrow \\ 4 \cdot 13 &= (-6)^2 + |D|v^2 && \Rightarrow \\ 52 &= 36 + |D|v^2 && \Leftrightarrow \\ 16 &= |D|v^2 \end{aligned}$$

This equation has two solutions for (D, v) , given by $(-4, \pm 2)$ and $(-16, \pm 1)$. Now looking at the first solution, we know that the case $D = -4$ implies $j = 1728$ and the constructed curve is defined by a Weierstraß equation XXX that has a vanishing parameter $b = 0$. We can therefore conclude that choosing $D = -4$ will not help us reconstructing TJJ_13 . It will produce curves with order 20, just not the one we are looking for.

So we choose the second solution $D = -16$ and in the next step, we have to compute the Hilbert class polynomial H_{-16} . To do so, we first have to find the set $ICG(D)$. To compute that set, observe, that since $\sqrt{\frac{|-16|}{3}} \approx 2.31 < 3$, we know from $A \leq \sqrt{\frac{|-16|}{3}}$ and $A \in \mathbb{Z}$ that A must be in the range $0..2$. So we loop through all possible values of A and through all possible values of B under the constraints $|B| \leq A$ and if $B < 0$ then $|B| < A$ and the compute potential C 's from $-16 = B^2 - 4AC$. We get the following two solution $(1, 0, 4)$ and $(2, 0, 2)$, giving we get

$$ICG(D) = \{(1, 0, 4), (2, 0, 2)\}$$

With the set $ICG(D)$ at hand we can compute the Hilbert class polynomial of $D = -16$. We can invoke `sagemath` to compute the j -invariant and get

$$\begin{aligned} H_{-16}(x) &= \left(x - j \left(\frac{i\sqrt{16}}{2} \right) \right) \left(x - j \left(\frac{i\sqrt{16}}{4} \right) \right) \\ &= (x - 287496)(x - 1728) \end{aligned}$$

So as we can see, in this particular case, the Hilbert class polynomial is a quadratic function with two integer corefficient. In the next step we have to project it onto a polynomial from

$\mathbb{F}_5[x]$, by computing the modular 5 remainder of the coefficients 1, 287496 and 1728. We get $287496 \bmod 13 = 1$ and $1728 \bmod 13 = 2$ from which follows that the projected Hilbert class polynomial is

$$H_{-11,5}(x) = (x-1)(x-12) = (x+12)(x+1)$$

3360 considered as a polynomial from $\mathbb{F}_5[x]$. So we have two roots given by $j = 1$ and $j = 12$. We al-
 3361 ready know that $j = 12$ is the wrong root to construct the tiny jubjub curve, since $1728 \bmod 13 =$
 3362 2 and that case can not construct a curve with $b \neq 0$. So we choose $j = 1$.

Another way to decide the proper root, is to compute the j -invariant of the tiny-jubjub curve. We get

$$\begin{aligned} j(TJJ_13) &= 12 \frac{4 \cdot 8^3}{4 \cdot 8^3 + 1 \cdot 8^2} \\ &= 12 \frac{4 \cdot 5}{4 \cdot 5 + 12} \\ &= 12 \frac{7}{7 + 12} \\ &= 12 \frac{7}{7 + 12} \\ &= 1 \end{aligned}$$

which is equal to the root $j = 1$ of the Hilbert class polynomial $H_{-16,13}$ as expected. We therefore have a situation with $j \neq 0$ and $j \neq 1728$, which tells us that we have to compute the parameter

$$c_1 = \frac{1}{12-1} = 6$$

in modular 5 arithmetics. Since $1728 \bmod 13 = 12$, we get $c_1 = 6$. Then we have to check if the curve $E(\mathbb{F}_5)$ defined by the Weierstraß $y^2 = x^3 + 3 \cdot 6x + 2 \cdot 6$ which is equivalent to

$$y^2 = x^3 + 5x + 12$$

has the correct order. We invoke sage and find that the order is 8, which implies that the trace of this curve is 6 not -6 as required. So we have to consider the second possibility and choose some quadratic non-residue $c_2 \in \mathbb{F}_{13}$. We choose $c_2 = 5$ and compute the Weierstraß equation $y^2 = x^3 + 5c_2^2 + 12c_2^3$ as

$$y^2 = x^3 + 8x + 5$$

We invoke sage and find that the order is 20, which is indeed the correct one. As we know from XXX, choosing any quadratic residue $d \in \mathbb{F}_5$ gives a curve of the same order defined by $y^2 = x^2 + ad^2x + bd^3$. Since 12 is a quadratic residue in \mathbb{F}_{13} , we can transform the curve defined by $y^2 = x^3 + 8x + 5$ into the curve $y^2 = x^3 + 12^2 \cdot 8 + 5 \cdot 12^3$ which gives

$$y^2 = x^3 + 8x + 8$$

3363 which is the ziny jubjub curve, that we used extensively throughout this book. So using the
 3364 complex multiplication method, we were able to derive a curve with specific properties from
 3365 scratch.

Example 102. To consider a real world example, we want to use the complex multiplication method in combination with sage to compute Secp256k1 from scratch. So by example XXX, we decided to compute an elliptic curve over a prime field \mathbb{F}_p of order r for the security parameters

$$\begin{aligned} p &= 115792089237316195423570985008687907853269984665640564039457584007908834671663 \\ r &= 115792089237316195423570985008687907852837564279074904382605163141518161494337 \end{aligned}$$

which, according to example XXX gives the trace of Frobenius $t = 432420386565659656852420866390673177327$. We also decided that we want a curve of the form $y^2 = x^3 + b$, that is we want the parameter a to be zero. This implies, the j -invariant of our curve must be zero.

In a first step we have to find a CM-discriminant D and some integer v , such that the equation

$$4p = t^2 + |D|v^2$$

is satisfied. Since we aim for a vanishing j -invariant, the first thing to try is $D = -3$. In this case we can compute $v^2 = (4p - t^2)$ and if v^2 happens to be an integer that has a square root v , we are done. Invoking sage we compute

```

sage: D = -3
sage: p = 1157920892373161954235709850086879078532699846656405
      64039457584007908834671663
sage: r = 1157920892373161954235709850086879078528375642790749
      04382605163141518161494337
sage: t = p+1-r
sage: v_sqr = (4*p - t^2)/abs(D)
sage: v_sqr.is_integer()
True
sage: v = sqrt(v_sqr)
sage: v.is_integer()
True
sage: 4*p == t^2 + abs(D)*v^2
True
sage: v
303414439467246543595250775667605759171

```

So indeed the pair $(D, v) = (-3, 303414439467246543595250775667605759171)$ solves the equation, which tells us that there is a curve of order r over a prime field of order p , defined by a Weierstraß equation $y^2 = x^3 + b$ for some $b \in \mathbb{F}_p$. So we need to compute b .

Now for $D = -3$ we already know that the associated Hilbert class polynomial is given by $H_{-3}(x) = x$, which gives the projected Hilbert class polynomial as $H_{-3,p} = x$ and the j -invariant of our curve is guaranteed to be $j = 0$. Now looking into table XXX, we see that there are 6 possible cases to construct a curve with the correct order r . In order to construct the curves of those case we have to choose some arbitrary quadratic and cubic non residue. So we loop through \mathbb{F}_p to find them, invoking sage:

```

sage: F = GF(p)
sage: for c2 in F:
.....:     try: # quadratic residue
.....:         _ = c2.nth_root(2)
.....:     except ValueError: # quadratic non residue
.....:         break
sage: c2
3
sage: for c3 in F:
.....:     try:
.....:         _ = c3.nth_root(3)
.....:     except ValueError:
.....:         break

```

```

3410 sage: c3
3411 2
525
526

3412 So we found the quadratic non residue  $c_2 = 3$  and the cubic non residue  $c_3 = 2$ . Using those
3413 numbers we check the six cases against the the expected order  $r$  of the curve we want to synte-
3414 size:

3415 sage: C1 = EllipticCurve(F, [0, 1])
3416 sage: C1.order() == r
3417 False
527
528
529
3418 sage: C2 = EllipticCurve(F, [0, c2^3])
3419 sage: C2.order() == r
3420 False
530
531
532
3421 sage: C3 = EllipticCurve(F, [0, c3^2])
3422 sage: C3.order() == r
3423 False
533
534
535
3424 sage: C4 = EllipticCurve(F, [0, c3^2*c2^3])
3425 sage: C4.order() == r
3426 False
536
537
538
3427 sage: C5 = EllipticCurve(F, [0, c3^(-2)])
3428 sage: C5.order() == r
3429 False
539
540
541
3430 sage: C6 = EllipticCurve(F, [0, c3^(-2)*c2^3])
3431 sage: C6.order() == r
3432 True
542
543
544

```

So as expeced we found an elliptic curve of the correct order r over a prime field of size p . So in principal we are done, as we have found a curve with the same basic properties as Secp256k1. However the curve is defined by the equation

$$y^2 = x^3 + 86844066927987146567678238756515930889952488499230423029593188005931626003754$$

that use a very large parameter b_1 , which might perform slow in certain algorithms. It is also not very elegant to be written down by hand. It might therefore be advantegous to find an isonmorphic curve with the smallest possible parameter b_2 . So in order to find such a b_2 , we have to choose a quadratic residue d , such that $b_2 = b_1 \cdot d^3$ is as small as possibl. To do so we rewrite the last equation into

$$d = \sqrt[3]{\frac{b_2}{b_1}}$$

```

3433 and then invoke sage to loop through values  $b_2 \in \mathbb{F}_p$  until it finds some number such that the
3434 quotient  $\frac{b_2}{b_1}$  has a cube root  $d$  and this cube root itself is a quadratic residue.

```

```

3435 sage: b1=86844066927987146567678238756515930889952488499230423
3436 029593188005931626003754
545
3437 sage: for b2 in F:
3438     ....:     try:
3439     ....:         d = (b2/b1).nth_root(3)
3440     ....:         try:
3441     ....:             _ = d.nth_root(2)
3442     ....:             if d != 0:
546
547
548
549
550
551

```

```

3443         ....:         break                               552
3444         ....:     except ValueError:                       553
3445         ....:         pass                                   554
3446         ....:     except ValueError:                       555
3447         ....:         pass                                   556
3448     sage: b2                                              557
3449         7                                                  558

```

So indeed the smallest possible value is $b_2 = 7$ and the defining Weierstrass equation of a curve over \mathbb{F}_p with prime order r is

$$y^2 = x^3 + 7$$

3450 which we might call secp256k1. As we have seen the complex multiplication method is power-
3451 ful enough to derive cryptographically secure curves like Secp256k1 from scratch.

3452 **The BLS6_6 pen& paper curve** In this paragraph we to summarize our understanding of
3453 elliptic curves to derive our main pen & paper example for the rest of the book. To do so, we
3454 want to use the complex multiplication method, to derive a pairing friendly elliptic curve that
3455 has similar properties to curves that are used in actual cryptographic protocols. However we
3456 design the curve specifically to be useful in pen&paper examples, which mostly means that the
3457 curve should contain only a few points, such that we are able to derive exhaustive addition and
3458 pairing tables.

3459 A well understood family of pairing friendly curves are the BLS curves (STUFF ABOUT
3460 THE HISTORY AND THE NAMING CONVENTION), which are derived in [XXX]. BLS
3461 curves are particular useful in our case if the embedding degree k satisfies $k \equiv 6 \pmod{0}$. Of
3462 course the smallest embedding degree k that satisfies this congruency, is $k = 6$ and we therefore
3463 aim for a BLS6 curve as our main pen&paper example.

3464 To apply the complex multiplication method from XXX, recall that this method starts with
3465 a definition of the base field \mathbb{F}_{p^m} as well as the trace of Frobenius t and the order of the curve.
3466 If the order $p^m + 1 - t$ is not a prime number, then what is necessary to control is the order r of
3467 the largest prime factor group.

In the case of BLS_6 curves, the parameter m is choosen to be 1, which means that the curves are defined over prime fields. All relevent parameters p , t and r are then themselves parameterized by the following three polynomials

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

3468 where Φ_6 is the 6-th cyclotomic polynomial and $x \in \mathbb{N}$ is a parameter that the designer has to
3469 choose in such a way that the evaluation of p , t and r at the point x gives integers that have
3470 the proper size to meet the security requirements of the curve that they want to design. It is
3471 then guaranteed that the complex multiplication method can be used in combination with those
3472 parameters to define an elliptic curve with CM-discriminant $D = -3$ and embedding degree
3473 $k = 6$ and curve equation $y^2 = x^3 + b$ for some $b \in \mathbb{F}_p$.

3474 For example if the curve should target the 128-bit security level, due to the Pholaard-rho
3475 attack (TODO) the parameter r shouls be prime number of at least 256 bits.

3476 In order to design the smallest, most unsecure BLS₆ curve, we therefore have to find a
 3477 parameter x , such that $r(x)$, $t(x)$ and $q(x)$ are the smallest natural numbers, that satisfy $q(x) > 3$
 3478 and $r(x) > 3$.

We therefore initiate the design process of our BLS₆ curve by looking-up the 6-th cyclotomic polynomial which is $\Phi_6 = x^2 - x + 1$ and then insert small values for x into the defining polynomials r, t, q . We get the following results:

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

3479 Since $q(1) = 1$ is not a prime number, the first x that gives a proper curve is $x = 2$. However
 3480 such a curve would be defined over a base field of characteristic 3 and we would rather like to
 3481 avoid that. We therefore find $x = 4$, which defines a curve over the prime field of characteristic
 3482 43, that has a trace of Frobenius $t = 5$ and a larger order prime group of size $r = 13$.

3483 Since the prime field \mathbb{F}_{43} has 43 elements and 43's binary representation is $43_2 = 101011$,
 3484 which are 6 digits, the name of our pen&paper curve should be BLS₆_6, since its is common
 3485 behaviour to name a BLS curve by its embedding degree and the bit-length of the modulus in
 3486 the base field. We call BLS₆_6 the **moon-math-curve**.

3487 Recalling from XXX, we know that the Hasse bound implies that BLS₆_6 will contain
 3488 exactly 39 elements. Since the prime factorization of 39 is $39 = 3 \cdot 13$, we have a "large" prime
 3489 factor group of size 13 as expected and a small cofactor group of size 3. Fortunately a subgroup
 3490 of order 13 is well suited for our purposes as 13 elements can be easily handled in the associated
 3491 addition, scalar multiplication and pairing tables in a pen and paper style.

3492 We can check that the embedding degree is indeed 6 as expected, since $k = 6$ is the smallest
 3493 number k such that $r = 13$ divides $43^k - 1$.

```

3494 sage: for k in range(1, 42): # Fermat's little theorem          559
3495     ....:     if (43^k-1)%13 == 0:                               560
3496     ....:         break                                           561
3497 sage: k                                                           562
3498 6                                                                  563
```

In order to compute the defining equation $y^2 = x^3 + ax + b$ of BLS₆-6, we use the complex multiplication method as described in XXX. The goal is to find $a, b \in \mathbb{F}_{43}$ representations, that are particularly nice to work with. The authors of XXX showed that the CM-discriminant of every BLS curve is $D = -3$ and indeed the equation

$$\begin{aligned} 4p &= t^2 + |D|v^2 && \Rightarrow \\ 4 \cdot 43 &= 5^2 + |D|v^2 && \Rightarrow \\ 172 &= 25 + |D|v^2 && \Leftrightarrow \\ 49 &= |D|v^2 \end{aligned}$$

has the four solutions $(D, v) \in \{(-3, -7), (-3, 7), (-49, -1), (-49, 1)\}$ if D is required to be negative, as expected. So $D = -3$ is indeed a proper CM-discriminant and we can deduce that the parameter a has to be 0 and that the Hilbert class polynomial is given by

$$H_{-3, 43}(x) = x$$

3499 This implies that the j -invariant of $BLS6_6$ is given by $j(BLS6_6) = 0$. We therefore have to
 3500 look at case XXX in table XXX to derive a parameter b . To decide the proper case for $j_0 = 0$
 3501 and $D = -3$, we therefore have to choose some arbitrary quadratic non residue c_2 and cubic
 3502 non residue c_3 in \mathbb{F}_{43} . We choose $c_2 = 5$ and $c_3 = 36$. We check

```

3503 sage: F43 = GF(43)                                     564
3504 sage: c2 = F43(5)                                       565
3505 .....: try: # quadratic residue                        566
3506 .....:     c2.nth_root(2)                               567
3507 .....: except ValueError: # quadratic non residue       568
3508 .....:     c2                                           569
3509 sage: c3 = F43(36)                                       570
3510 .....: try:                                             571
3511 .....:     c3.nth_root(3)                               572
3512 .....: except ValueError:                               573
3513 .....:     c3                                           574

```

3514 Using those numbers we check the six possible cases from XXX against the the expected
 3515 order 39 of the curve we want to synthesize:

```

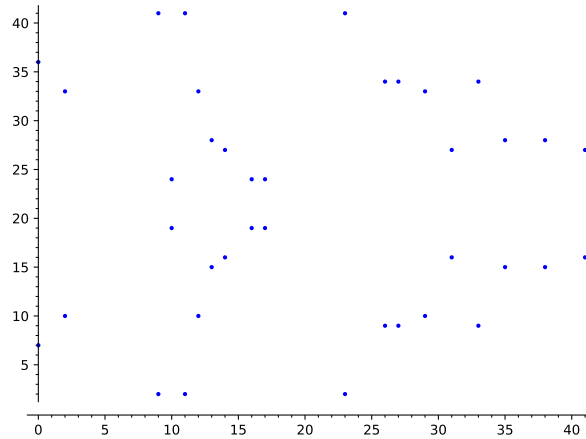
3516 sage: BLS61 = EllipticCurve(F43, [0, 1])               575
3517 sage: BLS61.order() == 39                               576
3518 False                                                  577
3519 sage: BLS62 = EllipticCurve(F43, [0, c2^3])             578
3520 sage: BLS62.order() == 39                               579
3521 False                                                  580
3522 sage: BLS63 = EllipticCurve(F43, [0, c3^2])             581
3523 sage: BLS63.order() == 39                               582
3524 True                                                  583
3525 sage: BLS64 = EllipticCurve(F43, [0, c3^2*c2^3])        584
3526 sage: BLS64.order() == 39                               585
3527 False                                                  586
3528 sage: BLS65 = EllipticCurve(F43, [0, c3^(-2)])          587
3529 sage: BLS65.order() == 39                               588
3530 False                                                  589
3531 sage: BLS66 = EllipticCurve(F43, [0, c3^(-2)*c2^3])     590
3532 sage: BLS66.order() == 39                               591
3533 False                                                  592
3534 sage: BLS6 = BLS63 # our BLS6 curve in the book        593

```

3535 So as expected we found an elliptic curve of the correct order 39 over a prime field of size 43,
 3536 defined by the equation

$$BLS6_6 := \{(x, y) \mid y^2 = x^3 + 6 \text{ for all } x, y \in \mathbb{F}_{43}\} \quad (5.26)$$

3537 There are other choice for b like $b = 10$ or $b = 23$, but all these curves are isomorphic and
 3538 hence represent the same curve really but in a different way only. Since BLS6-6 only contains
 3539 39 points it is possible to give a visual impression of the curve:



3540

3541 As we can see our curve is somewhat nice, as it does not contain self inverse points that is points
 3542 with $y = 0$. It follows that the addition law can be optimized, since the branch for those cases
 3543 can be eliminated.

3544 Summarizing the previous procedure, we have used the method of Barreto, Lynn and Scott
 3545 to construct a pairing friendly elliptic curve of embedding degree 6. However in order to do
 3546 elliptic curve cryptography on this curve note that since the order of $BLS6_6$ is 39 its group of
 3547 rational points is not a finite cyclic group of prime order. We therefore have to find a suitable
 3548 subgroup as our main target and since $39 = 13 \cdot 3$, we know that the curve must contain a "large"
 3549 prime order group of size 13 and a small cofactor group of order 3.

3550 It is the content of the following step to construct this group. One way to do so is to find
 3551 a generator. We can achieve this by choosing an arbitrary element of the group that is not the
 3552 point at infinity and then multiply that point with the cofactor of the groups order. If the result
 3553 is not the point at infinity, the result will be a generator and if it is the point at infinity we have
 3554 to choose a different element.

So in order to find a generator for the large order subgroup of size 13, we first notice that the cofactor of 13 is 3, since $39 = 3 \cdot 13$. We then need to construct an arbitrary element from $BLS6_6$. To do so in a pen and paper style, we can choose some *arbitrary* $x \in \mathbb{F}_{43}$ and see if there is some solution $y \in \mathbb{F}_{43}$ that satisfies the defining Weierstrass equation $y^2 = x^3 + 6$. We choose $x = 9$. Then $y = 2$ is a proper solution, since

$$\begin{aligned} y^2 &= x^3 + 6 && \Rightarrow \\ 2^2 &= 9^3 + 6 && \Leftrightarrow \\ 4 &= 4 \end{aligned}$$

3555 and this implies that $P = (9, 2)$ is therefore a point on $BLS6_6$. To see if we can project this
 3556 point onto a generator of the large order prim group $BLS6_6[13]$, we have to multiply P with
 3557 the cofactor, that is we have to compute $[3](9, 2)$. After some computation (EXERCISE) we get
 3558 $[3](9, 2) = (13, 15)$ and since this is not the point at infinity we know that $(13, 15)$ must be a
 3559 generator of $BLS6_6[13]$. We write

$$g_{BLS6_6[13]} = (13, 15) \tag{5.27}$$

3560 as we will need this generator in pairing computations all over the book. Since $g_{BLS6_6[13]}$
 3561 is a generator, we can use it to construct the subgroup $BLS6_6[13]$, by repeatedly adding the
 3562 generator to itself. We use sage and get

3563 **sage: P = BLS6(9, 2)**

594


```

3564 sage: Q = 3*P 595
3565 sage: Q.xy() 596
3566 (13, 15) 597
3567 sage: BLS6_13 = [] 598
3568 sage: for x in range(0,13): # cyclic of order 13 599
3569     P = x*Q 600
3570     BLS6_13.append(P) 601

```

Repeadly adding a generator to itself as we just did, will generate small groups in logarithmic order with respect to the generator as explained in XXX. We therefore get the following description of the large prime order subgroup of $BLS6_6$:

$$BLS6_6[13] = \{(13, 15) \rightarrow (33, 34) \rightarrow (38, 15) \rightarrow (35, 28) \rightarrow (26, 34) \rightarrow (27, 34) \rightarrow (27, 9) \rightarrow (26, 9) \rightarrow (35, 15) \rightarrow (38, 28) \rightarrow (33, 9) \rightarrow (13, 28) \rightarrow \mathcal{O}\} \quad (5.28)$$

Having a logarithmic description of this group is temendously helpfull in pen and paper computations. To see that, observe that we know from XXX that there is an exponential map from the scalar field \mathbb{F}_{13} to $BLS6_6[13]$ with respect to our generator

$$[\cdot]_{(13,15)} : \mathbb{F}_{13} \rightarrow BLS6_6[13] ; x \mapsto [x](13, 15)$$

which generates the group in logarithmic order. So for example we have $[1]_{(13,15)} = (13, 15)$, $[7]_{(13,15)} = (27, 9)$ and $[0]_{(13,15)} = \mathcal{O}$ and so on. The point for our purposes is, that we can use this representation to do computations in $BLS6_6[13]$ efficiently in our head using XXX. For example

$$\begin{aligned}
 (27, 34) \oplus (33, 9) &= [6](13, 15) \oplus [11](13, 15) \\
 &= [6 + 11](13, 15) \\
 &= [4](13, 15) \\
 &= (35, 28)
 \end{aligned}$$

So XXX is really all we need to do computations in $BLS6_6[13]$ in this book efficiently. However out of convinience, the following picture lists the entire addition table of that group. It might be useful in pen and paper computations:

\oplus	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)
\mathcal{O}	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)
(13, 15)	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}
(33, 34)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)
(38, 15)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)
(35, 28)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)
(26, 34)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)
(27, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)
(27, 9)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)
(26, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)
(35, 15)	(35, 15)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)
(38, 28)	(38, 28)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)
(33, 9)	(33, 9)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)
(13, 28)	(13, 28)	\mathcal{O}	(13, 15)	(33, 34)	(38, 15)	(35, 28)	(26, 34)	(27, 34)	(27, 9)	(26, 9)	(35, 15)	(38, 28)	(33, 9)

Now that we have constructed a "large" cyclic prime order subgroup of $BLS6_6$ suitable for many pen and paper computations in elliptic curve cryptography, we have to look at how to do

pairings in this context. We know that $BLS6_6$ is a pairing friendly curve by design, since it has a small embedding degree $k = 6$. It is therefore possible to compute Weil pairings efficiently. However in order to do so, we have to decide the groups \mathbb{G}_1 and \mathbb{G}_2 as explained in XXX.

Since $BLS6_6$ has two non trivial subgroups it would be possible to use any of them as the n -torsion group. However in cryptography the only secure choice is to use the large prime order subgroup, which in our case is $BLS6_6[13]$. we therefore decide to consider the 13-torsion and define

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

as the first argument for the Weil pairing function.

In order to construct the domain for the second argument, we need to construct $\mathbb{G}_2[13]$, which, according to the general theory should be defined by those elements P of the full 13-torsion group $BLS6_6[13]$, that are mapped to $43 \cdot P$ under the Frobenius endomorphism XXX.

To compute $\mathbb{G}_2[13]$ we therefore have to find the full 13-torsion group first. To do so, we use the technique from XXX, which tells us, that the full 13-torsion can be found in the curve extension

$$BLS6_6 := \{(x, y) \mid y^2 = x^3 + 6 \text{ for all } x, y \in \mathbb{F}_{43^6}\} \quad (5.29)$$

over the extension field \mathbb{F}_{43^6} , since the embedding degree of $BLS6_6$ is 6. So we have to construct \mathbb{F}_{43^6} , a field that contains 6321363049 many elements. In order to do so we use the procedure of XXX and start by choosing a non-reducible polynomial of degree 6 from the ring of polynomials $\mathbb{F}_{43}[t]$. We choose $p(t) = t^6 + 6$. Using sage we get

```
sage: F43 = GF(43)                                602
sage: F43t.<t> = F43[]                             603
sage: p = F43t(t^6+6)                             604
sage: p.is_irreducible()                          605
True                                              606
sage: F43_6.<v> = GF(43^6, name='v', modulus=p)    607
```

Recall from XXX that elements $x \in \mathbb{F}_{43^6}$ can be seen as polynomials $a_0 + a_1v + a_2v^2 + \dots + a_5v^5$ with the usual addition of polynomials and multiplication modulo $t^6 + 6$.

In order to compute $\mathbb{G}_2[13]$ we first have to extend $BLS6_6$ to \mathbb{F}_{43^6} , that is we keep the defining equation but extend the domain from \mathbb{F}_{43} to \mathbb{F}_{43^6} . After that we have to find at least one element P from that curve, that is not the point at infinity, that is in the full 13-torsion and that satisfies the identity $\pi(P) = [43]P$. We can then use this element as our generator of $\mathbb{G}_2[13]$ and construct all other elements by repeated addition to itself.

Since $BLS6(\mathbb{F}_{43^6})$ contains 6321251664 elements, its not a good strategy to simply loop through all elements. Fortunately sage has a way to loop through elements from the torsion group directly. We get

```
sage: BLS6 = EllipticCurve(F43_6, [0, 6]) # curve extension 608
sage: INF = BLS6(0) # point at infinity 609
sage: for P in INF.division_points(13): # full 13-torsion 610
.....: # PI(P) == [q]P 611
.....:     if P.order() == 13: # exclude point at infinity 612
.....:         PiP = BLS6([a.frobenius() for a in P]) 613
.....:         qP = 43*P 614
```

```

3610     ....:         if PiP == qP:                                615
3611     ....:             break                                     616
3612     sage: P.xy()                                              617
3613     (7*v^2, 16*v^3)                                           618

```

3614 So we found an element from the full 13-torsion, that is in the Eigenspace of the Eigenvalue 43,
 3615 which implies that it is an element of $\mathbb{G}_2[13]$. As $\mathbb{G}_2[13]$ is cyclic of prime order this element
 3616 must be a generator and we write

$$g_{\mathbb{G}_2[13]} = (7v^2, 16v^3) \quad (5.30)$$

3617 We can use this generator to compute \mathbb{G}_2 is logarithmic order with respect to $g_{\mathbb{G}_2[13]}$. Using
 3618 sage we get

```

3619 sage: Q = BLS6(7*v^2, 16*v^3)                                619
3620 sage: BLS6_13_2 = []                                         620
3621 sage: for x in range(0, 13):                                  621
3622     ....:     P = x*Q                                           622
3623     ....:     BLS6_13_2.append(P)                               623

```

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

Again, having a logarithmic description of $\mathbb{G}_2[13]$ is tremendously helpfull in pen and paper computations, as it reduces complicated computation in the extended curve to modular 13 arithmetics. For example

$$\begin{aligned} (17v^2, 28v^3) \oplus (10v^2, 15v^2) &= [6](7v^2, 16v^3) \oplus [11](7v^2, 16v^3) \\ &= [6 + 11](7v^2, 16v^3) \\ &= [4](7v^2, 16v^3) \\ &= (37v^2, 27v^3) \end{aligned}$$

3624 So XXX is really all we need to do computations in $\mathbb{G}_2[13]$ in this book efficiently.

To summerize the previous steps, we have found two subgroups $\mathbb{G}_1[13]$ as well as $\mathbb{G}_2[13]$ suitable to do Weil pairings on *BLS6_6* as explained in XXX. Using the logarithmic order XXX of $\mathbb{G}_1[13]$, the logarithmic order XXX of $\mathbb{G}_2[13]$ and the bilinearity

$$e([k_1]g_{BLS6_6[13]}, [k_2]g_{\mathbb{G}_2[13]}) = e(g_{BLS6_6[13]}, g_{\mathbb{G}_2[13]})^{k_1 \cdot k_2}$$

we can do Weil pairings on *BLS6_6* in a pen and paper style, observing that the Weil pairing between our two generators is given by the identity

$$e(g_{BLS6_6[13]}, g_{\mathbb{G}_2[13]}) = 5v^5 + 16v^4 + 16v^3 + 15v^2 + 3v + 41$$

3625

```

3626 sage: g1 = BLS6([13, 15])                                     624
3627 sage: g2 = BLS6([7*v^2, 16*v^3])                             625
3628 sage: g1.weil_pairing(g2, 13)                                 626
3629 5*v^5 + 16*v^4 + 16*v^3 + 15*v^2 + 3*v + 41                 627

```

3630 **Hashing to the pairing groups** We give various constructions to hash into \mathbb{G}_1 and \mathbb{G}_2 .

3631 We start with hashing to the scalar field... TO APPEAR

3632 Non of these techniques work for hashing into \mathbb{G}_2 . We therefore implement Pederson's
3633 Hash for BLS6.

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

$$H_1 : \{0, 1\}^{12} \rightarrow \mathbb{G}_1$$

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:

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

Note that $a^x = 1$ 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

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

The hash of 011010101100 is given by

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

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

$$H_2 : \{0, 1\}^{12} \rightarrow \mathbb{G}_2$$

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:

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

We extend this to a hash function that maps unbounded bitstring to \mathbb{G}_2 by precomposing with an actual hash function like *MD5* and feed the first 12 bits of its outcome into our previously defined hash function.

$$TinyMD5_{\mathbb{G}_2} : \{0, 1\}^* \rightarrow \mathbb{G}_2$$

3634 with $TinyMD5_{\mathbb{G}_2}(s) = H_2(MD5(s)_0, \dots, MD5(s)_{11})$. For example, since $MD5("") = 0xd41d8cd98f00b204e98$
 3635 and the binary representation of the hexadecimal number $0x27e$ is 001001111110 we compute
 3636 $TinyMD5_{\mathbb{G}_2}$ of the empty string as $TinyMD5_{\mathbb{G}_2}("") = H_2(MD5(s)_{11}, \dots, MD5(s)_0) = H_2(001001111110) =$

Chapter 6

Statements

As we have seen in the informal introduction XXX, a SNARK is a short non-interactive argument of knowledge, where the knowledge-proof attests to the correctness of statements like “The prover knows the prime factorization of a given number” or “The prover knows the preimage to a given SHA2 digest value” and similar things. However, human readable statements like these are imprecise and not very useful from a formal perspective.

Chapter 1?

In this chapter we therefore look more closely at ways to formalize statements in mathematically rigorous ways, useful for SNARK development. We start by introducing formal languages as a way to define statements properly (section 6.1). We will then look at algebraic circuits and rank-1 constraint systems as two particularly useful ways to define statements in certain formal languages (section 6.2). After that, we will have a look at fundamental building blocks of compilers that compile high-level languages to circuits and associated rank-1 constraint systems.

Proper statement design should be of high priority in the development of SNARKs, since unintended true statements can lead to potentially severe and almost undetectable security vulnerabilities in the applications of SNARKs.

6.1 Formal Languages

Formal languages provide the theoretical background in which statements can be formulated in a logically rigorous way and where proving the correctness of any given statement can be realized by computing words in that language.

"rigorous"?

One might argue that the understanding of formal languages is not very important in SNARK development and associated statement design, but terms from that field of research are standard jargon in many papers on zero-knowledge proofs. We therefore believe that at least some introduction to formal languages and how they fit into the picture of SNARK development is beneficial, mostly to give developers a better intuition about where all this is located in the bigger picture of the logic landscape. In addition, formal languages give a better understanding of what a formal proof for a statement actually is.

"proving"?

Roughly speaking, a formal language (or just language for short) is nothing but a set of words, *th*. Words, in turn, are strings of letters taken from some alphabet and formed according to some defining rules of the language.

To be more precise, let Σ be any set and Σ^* the set of all finite **tuples** (ordered lists) (x_1, \dots, x_n) of elements x_j from Σ including the empty tuple $() \in \Sigma^*$. Then, a **language** L , in its most general definition, is nothing but a subset of Σ^* . In this context, the set Σ is called the **alphabet** of the language L , elements from Σ are called letters and elements from L are called **words**. The rules that specify which tuples from Σ^* belong to the language and which don't,

are called the **grammar** of the language. S: I suggest adding an example based on English, e.g. “tea” and “eat” are words of English, but “aet” and “tae” are not

Add ex-ample

If L_1 and L_2 are two formal languages over the same alphabet, we call L_1 and L_2 **equivalent**, if there is a 1:1 correspondence between the words in L_1 and the words in L_2 . S: I’d add “In other words, two languages are equivalent if they generate the same set of words.”

Add more explanation

Decision Functions Our previous definition of formal languages is very general and many subclasses of languages like **regular languages** or **context-free languages** are known in the literature. However, in the context of SNARK development, languages are commonly defined as **decision problems** where a so-called **deciding relation** $R \subset \Sigma^*$ decides whether a given tuple $x \in \Sigma^*$ is a word in the language or not. If $x \in R$ then x is a word in the associated language L_R and if $x \notin R$ then not. The relation R therefore summarizes the grammar of language L_R .

I’d delete this, too distracting

Unfortunately, in some literature on proof systems, $x \in R$ is often written as $R(x)$, which is misleading since in general R is not a function but a relation in Σ^* . For the sake of this book we therefore adopt a different point of view and work with what we might call a **decision function** instead:

$$R : \Sigma^* \rightarrow \{true, false\} \quad (6.1)$$

Decision functions therefore decide if a tuple $x \in \Sigma^*$ is an element of a language or not. In case a decision function is given, the associated language itself can be written as the set of all tuples that are decided by R , i.e as the set:

$$L_R := \{x \in \Sigma^* \mid R(x) = true\} \quad (6.2)$$

In the context of formal languages and decision problems, a **statement** S is the claim that language L contains a word x , i.e a statement claims that there exist some $x \in L$. A constructive **proof** for statement S is given by some string $P \in \Sigma^*$ and such a proof is **verified** by checking $R(P) = true$. In this case, P is called an **instance** of the statement S .

While the term **language** might suggest a deeper relation to the well known **natural languages** like English, formal languages and natural languages differ in many ways. The following examples will provide some intuition about formal languages, highlighting the concepts of statements, proofs and instances:

Example 103 (Alternating Binary strings). To consider a very basic formal language with an almost trivial grammar, consider the set $\{0, 1\}$ of the two letters 0 and 1 as our alphabet Σ and imply the rule that a proper word must consist of alternating binary letters of arbitrary length.

Then, the associated language L_{alt} is the set of all finite binary tuples, where a 1 must follow a 0 and vice versa. So, for example, $(1, 0, 1, 0, 1, 0, 1, 0, 1) \in L_{alt}$ is a proper word in this languages as is $(0) \in L_{alt}$ or the empty word $() \in L_{alt}$. However, the binary tuple $(1, 0, 1, 0, 1, 0, 1, 1, 1) \in \{0, 1\}^*$ is not a proper word, as it violates the grammar of L_{alt} : the last3 letters are all 1. Furthermore, the tuple $(0, A, 0, A, 0, A, 0)$ is not a proper word, as not all its letters are not from the proper alphabet: we defined the alphabet Σ as the set $\{0, 1\}$, and A is not part of that set.

Attempting to write the grammar of this language in a more formal way, we can define the following decision function:

$$R : \{0, 1\}^* \rightarrow \{true, false\} ; (x_0, x_1, \dots, x_n) \mapsto \begin{cases} true & x_{j-1} \neq x_j \text{ for all } 1 \leq j \leq n \\ false & \text{else} \end{cases}$$

We can use this function to decide if arbitrary **binary tuples** are words in L_{alt} or not. Some examples:

binary tuples

- 3710 • $R(1, 0, 1) = \text{true}$,
- 3711 • $R(0) = \text{true}$,
- 3712 • $R() = \text{true}$,
- 3713 • but $R(1, 1) = \text{false}$.

3714 Inside our language L_{alt} , it makes sense to claim the following statement: “There exists an
 3715 alternating string.” One way to prove this statement constructively is by providing an actual
 3716 instance, that is, finding actual alternating string like $x = (1, 0, 1)$. Constructing string $(1, 0, 1)$
 3717 therefore proves the statement “There exists an alternating string.”, because it is easy to verify
 3718 that $R(1, 0, 1) = \text{true}$.

3719 *Example 104 (Programming Language).* Programming languages are a very important class of
 3720 formal languages. For these languages, the alphabet is usually (a subset) of the ASCII table,
 3721 and the grammar is defined by the rules of the programming language’s compiler. Words, then,
 3722 are nothing but properly written computer programs that the compiler accepts. The compiler
 3723 can therefore be interpreted as the decision function.

3724 To give an unusual example strange enough to highlight the point, consider the program-
 3725 ming language Malbolge as defined in XXX. This language was specifically designed to be
 3726 almost impossible to use and writing programs in this language is a difficult task. An inter-
 3727 esting claim is therefore the statement: “There exists a computer program in Malbolge”. As it
 3728 turned out, proving this statement constructively, that is, by providing an actual instance of such
 3729 a program, was not an easy task, as it took two years after the introduction of Malbolge to write
 3730 a program that its compiler accepts. So, for two years, no one was able to prove the statement
 3731 constructively.

add refer-
ence

To look at this high-level description more formally, we write $L_{Malbolge}$ for the language that
 uses the ASCII table as its alphabet and its words are tuples of ASCII letters that the Malbolge
 compiler accepts. Proving the statement “There exists a computer program in Malbolge” is then
 equivalent to the task of finding some word $x \in L_{Malbolge}$. The string

$(=<\#9] 6ZY327Uv4-QsqpMn\&+Ij''E\%e\{Ab w=_:]Kw\%o44Uqp0/Q?xNvL:'H\%c\#DD2\wedge WV>gY;dts76qKJImZkj$

3732 is an example of such a proof, as it is excepted by the Malbolge compiler and is compiled to
 3733 an executable binary that displays “Hello, World.” (See XXX). In this example, the Malbolge
 3734 compiler therefore serves as the verification process.

add refer-
ence

Example 105 (The Empty Language). To see that not every language has even one word, con-
 sider the alphabet $\Sigma = \mathbb{Z}_6$, where \mathbb{Z}_6 is the ring of modular 6 arithmetics as derived in example
 8 in chapter 3, together with the following decision function

check
reference

$$R_\emptyset : \mathbb{Z}_6^* \rightarrow \{\text{true}, \text{false}\} ; (x_1, \dots, x_n) \mapsto \begin{cases} \text{true} & n = 4 \text{ and } x_1 \cdot x_1 = 2 \\ \text{true} & \text{else} \end{cases}$$

3735 We write L_\emptyset for the associated language. As we can see from the multiplication table of \mathbb{Z}_6
 3736 in example 8 in chapter 3, the ring \mathbb{Z}_6 does not contain any element x such that $x^2 = 2$, which
 3737 implies $R_\emptyset(x_1, \dots, x_n) = \text{false}$ for all tuples $(x_1, \dots, x_n) \in \Sigma^*$. The language therefore does
 3738 not contain any words. Proving the statement “There exists a word in L_\emptyset ” constructively by
 3739 providing an instance is therefore impossible. The verification will never check any tuple.

check
reference

Example 106 (3-Factorization). We will use the following simple example repeatedly throughout this book. The task is to develop a SNARK that proves knowledge of three factors of an element from the finite field \mathbb{F}_{13} . There is nothing particularly useful about this example from an application point of view, however, in a sense, it is the most simple example that gives rise to a non trivial SNARK in some of the most common zero-knowledge proof systems.

Formalizing the high-level description, we use $\Sigma := \mathbb{F}_{13}$ as the underlying alphabet of this problem and define the language $L_{3.fac}$ to consists of those tuples of field elements from \mathbb{F}_{13} that contain exactly 4 letters w_1, w_2, w_3, w_4 which satisfy the equation $w_1 \cdot w_2 \cdot w_3 = w_4$.

So, for example, the tuple $(2, 12, 4, 5)$ is a word in $L_{3.fac}$, while neither $(2, 12, 11)$, nor $(2, 12, 4, 7)$ nor $(2, 12, 7, 168)$ are words in $L_{3.fac}$ as they don't satisfy the grammar or are not define over the proper alphabet.

We can describe the language $L_{3.fac}$ more formally by introducing a decision function (as described in equation 6.1):

$$R_{3.fac} : \mathbb{F}_{13}^* \rightarrow \{true, false\} ; (x_1, \dots, x_n) \mapsto \begin{cases} true & n = 4 \text{ and } x_1 \cdot x_2 \cdot x_3 = x_4 \\ false & else \end{cases}$$

Having defined the language $L_{3.fac}$, it then makes sense to claim the statement "There is a word in $L_{3.fac}$ ". The way $L_{3.fac}$ is designed, this statement is equivalent to the statement "There are four elements w_1, w_2, w_3, w_4 from the finite field \mathbb{F}_{13} such that the equation $w_1 \cdot w_2 \cdot w_3 = w_4$ holds."

Proving the correctness of this statement constructively means to actually find some concrete field elements like $x_1 = 2, x_2 = 12, x_3 = 4$ and $x_4 = 5$ that satisfy the relation $R_{3.fac}$. The tuple $(2, 12, 4, 5)$ is therefore a constructive proof for the statement and the computation $R_{3.fac}(2, 12, 4, 5) = true$ is a verification of that proof. In contrast, the tuple $(2, 12, 4, 7)$ is not a proof of the statement, since the check $R_{3.fac}(2, 12, 4, 7) = false$ does not verify the proof.

Example 107 (Tiny JubJub Membership). In our main example, we derive a SNARK that proves a pair (x, y) of field elements from \mathbb{F}_{13} to be a point on the tiny jubjub curve in its Edwards form XXX.

In the first step, we define a language such that points on the tiny jubjub curve are in 1:1 correspondence with words in that language.

Since the tiny jubjub curve is an elliptic curve over the field \mathbb{F}_{13} , we choose the alphabet $\Sigma = \mathbb{F}_{13}$. In this case, the set \mathbb{F}_{13}^* consists of all finite strings of field elements from \mathbb{F}_{13} . To define the grammar, recall from XXX that a point on the tiny jubjub curve is a pair (x, y) of field elements such that $3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$. We can use this equation to derive the following decision function:

$$R_{tiny.jj} : \mathbb{F}_{13}^* \rightarrow \{true, false\} ; (x_1, \dots, x_n) \mapsto \begin{cases} true & n = 2 \text{ and } 3 \cdot x_1^2 + x_2^2 = 1 + 8 \cdot x_1^2 \cdot x_2^2 \\ false & else \end{cases}$$

The associated language $L_{tiny.jj}$ is then given as the set of all strings from \mathbb{F}_{13}^* that are mapped onto *true* by $R_{tiny.jj}$. We get

$$L_{tiny.jj} = \{(x_1, \dots, x_n) \in \mathbb{F}_{13}^* \mid R_{tiny.jj}(x_1, \dots, x_n) = true\}$$

We can claim the statement "There is a word in $L_{tiny.jj}$ " and because $L_{tiny.jj}$ is defined by $R_{tiny.jj}$, this statement is equivalent to the claim "The tiny jubjub curve in its Edwards form has curve a point."

A constructive proof for this statement is a pair (x, y) of field elements that satisfies the Edwards equation. Example XXX therefore implies that the tuple $(11, 6)$ is a constructive proof

Are we using w and x interchangeably or is there a difference between them?

check reference

jubjub

Edwards form

add reference

add reference

check wording

add reference

and the computation $R_{tiny.jj}(11,6) = true$ is a proof verification. In contrast, the tuple $(1,1)$ is not a proof of the statement, since the check $R_{tiny.jj}(1,1) = false$ does not verify the proof.

Exercise 39. Consider exercise XXX again. Define a decision function such that the associated language $L_{Exercise_{XXX}}$ consist precisely of all solutions to the equation $5x + 4 = 28 + 2x$ over \mathbb{F}_{13} . Provide a constructive proof for the claim: “There exist a word in $L_{Exercise_{XXX}}$ and verify the proof.

Exercise 40. Consider the modular 6 arithmetics \mathbb{Z}_6 from example 8 in chapter 3, the alphabet $\Sigma = \mathbb{Z}_6$ and the decision function

$$R_{example_8} : \Sigma^* \rightarrow \{true, false\} ; x \mapsto \begin{cases} true & x.len() = 1 \text{ and } 3 \cdot x + 3 = 0 \\ false & \text{else} \end{cases}$$

Compute all words in the associated language $L_{example_8}$, provide a constructive proof for the statement “There exist a word in $L_{example_example_8}$ ” and verify the proof.

check
references

Instance and Witness As we have seen in the previous paragraph, statements provide membership claims in formal languages, and instances serve as constructive proofs for those claims. However, in the context of **zero-knowledge** proof systems, our naive notion of constructive proofs is refined in such a way that its possible to hide parts of the proof instance and still be able to prove the statement. In this context, it is therefore necessary to split a proof into a **public part** called the **instance** and a private part called a **witness**.

To account for this separation of a proof instance into a public and a private part, our previous definition of formal languages needs a refinement in the context of zero-knowledge proof systems. Instead of a single alphabet, the refined definition considers two alphabets Σ_I and Σ_W , and a decision function defined as follows:

$$R : \Sigma_I^* \times \Sigma_W^* \rightarrow \{true, false\} ; (i; w) \mapsto R(i; w) \quad (6.3)$$

Words are therefore tuples $(i; w) \in \Sigma_I^* \times \Sigma_W^*$ with $R(i; w) = true$. The refined definition differentiates between public inputs $i \in \Sigma_I$ and private inputs $w \in \Sigma_W$. The public input i is called an **instance** and the private input w is called a **witness** of R .

If a decision function is given, the associated language is defined as the set of all tuples from the underlying alphabet that are verified by the decision function:

$$L_R := \{(i; w) \in \Sigma_I^* \times \Sigma_W^* \mid R(i; w) = true\} \quad (6.4)$$

In this refined context, a **statement** S is a claim that, given an instance $i \in \Sigma_I^*$, there is a witness $w \in \Sigma_W^*$ such that language L contains a word $(i; w)$. A constructive **proof** for statement S is given by some string $P = (i; w) \in \Sigma_I^* \times \Sigma_W^*$ and a proof is **verified** by checking $R(P) = true$.

It is worth understanding the difference between statements as defined in XXX and the refined notion of statements from this paragraph. While statements in the sense of the previous paragraph can be seen as membership claims, statements in the refined definition can be seen as knowledge-proofs, where a prover claims knowledge of a witness for a given instance. For a more detailed discussion on this topic see [XXX sec 1.4]

add refer-
ence

add refer-
ence

Example 108 (SHA256 – Knowledge of Preimage). One of the most common examples in the context of zero-knowledge proof systems is the knowledge-of-a-preimage proof for some cryptographic hash function like *SHA256*, where a publicly known *SHA256* digest value is given,

and the task is to prove knowledge of a preimage for that digest under the *SHA256* function, without revealing that preimage.

preimage

To understand this problem in detail, we have to introduce a language able to describe the knowledge-of-preimage problem in such a way that the claim “Given digest i , there is a preimage w such that $SHA256(w) = i$ ” becomes a statement in that language. Since *SHA256* is a function

$$SHA256 : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$$

that maps binary strings of arbitrary length onto binary strings of length 256 and we want to prove knowledge of preimages, we have to consider binary strings of size 256 as instances and binary strings of arbitrary length as witnesses.

An appropriate alphabet Σ_I for the set of all instances and an appropriate alphabet Σ_W for the set of all witnesses is therefore given by the set $\{0, 1\}$ of the two binary letters and a proper decision function is given by:

$$R_{SHA256} : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{true, false\} ;$$

$$(i; w) \mapsto \begin{cases} true & i.len() = 256, i = SHA256(w) \\ false & else \end{cases}$$

We write L_{SHA256} for the associated language and note that it consists of words, which are tuples $(i; w)$ such that the instance i is the *SHA256* image of the witness w .

Given some instance $i \in \{0, 1\}^{256}$, a statement in L_{SHA256} is the claim “Given digest i , there is a preimage w such that $SHA256(w) = i$ ”, which is exactly what the knowledge-of-preimage problem is about. A constructive proof for this statement is therefore given by a preimage w to the digest i and proof verification is achieved by checking that $SHA256(w) = i$.

Example 109 (3-factorization). To give an intuition about the implication of refined languages, consider $L_{3, fac}$ from example 106 again. As we have seen, a constructive proof in $L_{3, fac}$ is given by 4 field elements x_1, x_2, x_3 and x_4 from \mathbb{F}_{13} such that the product in modular 13 arithmetics of the first three elements is equal to the 4'th element.

check reference

Splitting words from $L_{3, fac}$ into private and public parts, we can reformulate the problem and introduce different levels of privacy into the problem. For example, we could reformulate the membership statement of $L_{3, fac}$ into a statement where all factors x_1, x_2, x_3 of x_4 are private and only the product x_4 is public. A statement for this reformulation is then expressed by the claim: “Given a publicly known field element x_4 , there are three private factors of x_4 ”. Assuming some instance x_4 , a constructive proof for the associated knowledge claim is then provided by any tuple (x_1, x_2, x_3) such that $x_1 \cdot x_2 \cdot x_3 = x_4$.

At this point, it is important to note that, while constructive proofs in the refinement don't look very different from constructive proofs in the original language, we will see in XXX that there are proof systems able to prove the statement (at least with high probability) without revealing anything about the factors x_1, x_2 , or x_3 . This is why the importance of the refinement only becomes clear once more elaborate proofing methods beyond naive constructive proofs are provided.

add reference

We can formalize this new language, which we might call L_{3, fac_zk} , by defining the following decision function:

$$R_{3, fac_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\} ;$$

$$((i_1, \dots, i_n); (w_1, \dots, w_m)) \mapsto \begin{cases} true & n = 1, m = 3, i_1 = w_1 \cdot w_2 \cdot w_3 \\ false & else \end{cases}$$

3833 The associated language $L_{3.fac_zk}$ is defined by all tuples from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto
 3834 *true* under the decision function $R_{3.fac_zk}$.

3835 Considering the distinction we made between the instance and the witness part in $L_{3.fac_zk}$,
 3836 one might ask why we chose the factors x_1, x_2 and x_3 to be the witness and the product x_4 to
 3837 be the instance and why we didn't choose another combination? This was an arbitrary choice
 3838 in the example. Every other combination of private and public factors would be equally valid.
 3839 For example, it would be possible to declare all variables as private or to declare all variables as
 3840 public. Actual choices are determined by the application only.

3841 *Example 110 (The Tiny JubJub Curve).* Consider the language $L_{tiny.jj}$ from example 107. As
 3842 we have seen, a constructive proof in $L_{tiny.jj}$ is given by a pair (x_1, x_2) of field elements from
 3843 \mathbb{F}_{13} such that the pair is a point of the tiny jubjub curve in its Edwards representation.

check
reference

3844 We look at a reasonable splitting of words from $L_{tiny.jj}$ into private and public parts. The
 3845 two obvious choices are to either choose both coordinates x_1 as x_2 as public inputs, or to choose
 3846 both coordinates x_1 as x_2 as private inputs.

In case both coordinates are public, we define the grammar of the associated language by introducing the following decision function:

$$R_{tiny.jj.1} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\};$$

$$(I_1, \dots, I_n; W_1, \dots, W_m) \mapsto \begin{cases} true & n = 2, m = 0 \text{ and } 3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2 \\ false & \text{else} \end{cases}$$

3847 The language $L_{tiny.jj.1}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto
 3848 *true* by $R_{tiny.jj.1}$.

In case both coordinates are private, we define the grammar of the associated refined language by introducing the following decision function:

$$R_{tiny.jj_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\};$$

$$(I_1, \dots, I_n; W_1, \dots, W_m) \mapsto \begin{cases} true & n = 0, m = m \text{ and } 3 \cdot W_1^2 + W_2^2 = 1 + 8 \cdot W_1^2 \cdot W_2^2 \\ false & \text{else} \end{cases}$$

3849 The language $L_{tiny.jj_zk}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto
 3850 *true* by $R_{tiny.jj_zk}$.

Exercise 41. Consider the modular 6 arithmetics \mathbb{Z}_6 from example 8 in chapter 3 as alphabets Σ_I and Σ_W and the following decision function

check
reference

$$R_{linear} : \Sigma^* \times \Sigma^* \rightarrow \{true, false\};$$

$$(i; w) \mapsto \begin{cases} true & i.len() = 3 \text{ and } w.len() = 1 \text{ and } i_1 \cdot w_1 + i_2 = i_3 \\ false & \text{else} \end{cases}$$

3851 Which of the following instances (i_1, i_2, i_3) has a proof of knowledge in L_{linear} ?

- 3852 • $(3, 3, 0)$
- 3853 • $(2, 1, 0)$
- 3854 • $(4, 4, 2)$

Exercise 42 (Edwards Addition on Tiny JubJub). Consider the tiny-jubjub curve together with its Edwards addition law from example XXX. Define an instance alphabet Σ_I , a witness alphabet Σ_W and a decision function R_{add} with associated language L_{add} such that a string $(i; w) \in \Sigma_I^* \times \Sigma_W^*$ is a word in L_{add} if and only if i is a pair of curve points on the tiny-jubjub curve in Edwards form and w is the Edwards sum of those curve points.

Choose some instance $i \in \Sigma_I^*$, provide a constructive proof for the statement “There is a witness $w \in \Sigma_W^*$ such that $(i; w)$ is a word in L_{add} ” and verify that proof. Then find some instance $i \in \Sigma_I^*$ such that i has no knowledge proof in L_{add} .

Modularity From a developers perspective, it is often useful to construct complex statements and their representing languages from simple ones. In the context of zero-knowledge proof systems, those simple building blocks are often called **gadgets**, and gadget libraries usually contain representations of atomic types like booleans, integers, various hash functions, elliptic curve cryptography and many more. In order to synthesize statements, developers then combine predefined gadgets into complex logic. We call the ability to combine statements into more complex statements **modularity**.

To understand the concept of modularity on the level of formal languages defined by decision functions, we need to look at the **intersection** of two languages, which exists whenever both languages are defined over the same alphabet. In this case, the intersection is a language that consists of strings which are words in both languages.

To be more precise, let L_1 and L_2 be two languages defined over the same instance and witness alphabets Σ_I and Σ_W . Then the intersection $L_1 \cap L_2$ of L_1 and L_2 is defined as

$$L_1 \cap L_2 := \{x \mid x \in L_1 \text{ and } x \in L_2\} \quad (6.5)$$

If both languages are defined by decision functions R_1 and R_2 , the following function is a decision function for the intersection language $L_1 \cap L_2$:

$$R_{L_1 \cap L_2} : \Sigma_I^* \times \Sigma_W^* \rightarrow \{true, false\}; (i, w) \mapsto R_1(i, w) \text{ and } R_2(i, w) \quad (6.6)$$

The fact that the intersection of two decision function based languages is a decision function based language again is important from an implementations point of view: it allows to construct complex decision functions, their languages and associated statements from simple building blocks. Given a publicly known instance $i \in \Sigma_I^*$ a statement in an intersection language then claims knowledge of a witness that satisfies all relations simultaneously.

6.2 Statement Representations

As we have seen in the previous section, formal languages and their definitions by decision functions are a powerful tool to describe statements in a formally rigorous manner.

However, from the perspective of existing zero-knowledge proof systems, not all ways to actually represent decision functions are equally useful. Depending on the proof system, some are more suitable than others. In this section, will describe two of the most common ways to represent decision functions and their statements.

6.2.1 Rank-1 Quadratic Constraint Systems

Although decision functions are expressible in various ways, many contemporary proofing systems require the deciding relation to be expressed in terms of a system of quadratic equations

add reference

Can we reword this? It's grammatically correct but hard to read

over a finite field. This is true in particular for pairing-based proofing systems like XXX, roughly because it is possible to check solutions to those equations “in the exponent” of pairing-friendly cryptographic groups.

add reference

In this section, we will therefore have a closer look at a particular type of quadratic equation called **rank-1 quadratic constraints systems**, which are a common standard in zero-knowledge proof systems. We will start with a general introduction to those systems and then look at their relation to formal languages. We will look into a common way to compute solutions to those systems, and then show how a simple compiler might derive rank-1 constraint systems from more high-level programming code.

R1CS representation To understand what **rank-1 (quadratic) constraint systems** are in detail, let \mathbb{F} be a field, n, m and $k \in \mathbb{N}$ three numbers and a_j^i, b_j^i and $c_j^i \in \mathbb{F}$ constants from \mathbb{F} for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$. Then a rank-1 constraint system (R1CS) is defined as follows:

$$\begin{aligned} (a_0^1 + \sum_{j=1}^n a_j^1 \cdot I_j + \sum_{j=1}^m a_{n+j}^1 \cdot W_j) \cdot (b_0^1 + \sum_{j=1}^n b_j^1 \cdot I_j + \sum_{j=1}^m b_{n+j}^1 \cdot W_j) &= c_0^1 + \sum_{j=1}^n c_j^1 \cdot I_j + \sum_{j=1}^m c_{n+j}^1 \cdot W_j \\ &\vdots \\ (a_0^k + \sum_{j=1}^n a_j^k \cdot I_j + \sum_{j=1}^m a_{n+j}^k \cdot W_j) \cdot (b_0^k + \sum_{j=1}^n b_j^k \cdot I_j + \sum_{j=1}^m b_{n+j}^k \cdot W_j) &= c_0^k + \sum_{j=1}^n c_j^k \cdot I_j + \sum_{j=1}^m c_{n+j}^k \cdot W_j \end{aligned}$$

If a rank-1 constraint system is given, the parameter k is called the **number of constraints**. If a tuple $(I_1, \dots, I_n; W_1, \dots, W_m)$ of field elements satisfies these equations, (I_1, \dots, I_n) is called an **instance** and (W_1, \dots, W_m) is called an associated **witness** of the system.

Remark 1 (Matrix notation). The presentation of rank-1 constraint systems can be simplified using the notation of vectors and matrices, which abstracts over the indices. In fact if $x = (1, I, W) \in \mathbb{F}^{1+n+m}$ is a $(n+m+1)$ -dimensional vector, A, B, C are $(n+m+1) \times k$ -dimensional matrices and \odot is the **Schur/Hadamard product**, then a R1CS can be written as

Schur/Hadamard product

$$Ax \odot Bx = Cx$$

However, since we did not introduce matrix calculus in the book, we use XXX as the defining equation for rank-1 constraints systems. We only highlighted the matrix notation, because it is sometimes used in the literature.

add reference

Generally speaking, the idea of a rank-1 constraint system is to keep track of all the values that any variable can assume during a computation and to bind the relationships among all those variables that are implied by the computation itself. Enforcing relations between all the steps of a computer program, the execution is then constrained to be computed in exactly the expected way without any opportunity for deviations. In this sense, solutions to rank-1 constraint systems are proofs of proper program execution.

Example 111 (3-Factorization). To provide a better intuition of rank-1 constraint systems, consider the language L_{3, fac_zk} from example 106 again. As we have seen, L_{3, fac_zk} consists of words $(I_1; W_1, W_2, W_3)$ over the alphabet \mathbb{F}_{13} such that $I_1 = W_1 \cdot W_2 \cdot W_3$. We show how to rewrite the decision function as a rank-1 constraint system.

check reference

Since R1CS are systems of quadratic equations, expressions like $W_1 \cdot W_2 \cdot W_3$ which contain products of more than two factors (which are therefore not quadratic) have to be rewritten in a process often called **flattening**. To flatten the defining equation $I_1 = W_1 \cdot W_2 \cdot W_3$ of L_{3, fac_zk} we introduce a new variable W_4 , which captures two of the three multiplications in $W_1 \cdot W_2 \cdot W_3$. We

get the following two constraints

$$W_1 \cdot W_2 = W_4 \quad \text{constraint 1}$$

$$W_4 \cdot W_3 = I_1 \quad \text{constraint 2}$$

3918 Given some instance I_1 , any solution (W_1, W_2, W_3, W_4) to this system of equations provides a
 3919 solution to the original equation $I_1 = W_1 \cdot W_2 \cdot W_3$ and vice versa. Both equations are therefore
 3920 equivalent in the sense that solutions are in a 1:1 correspondence.

3921 Looking at both equations, we see how each constraint enforces a step in the computation.
 3922 In fact, the first constraint forces any computation to multiply the witness W_1 and W_2 first. Oth-
 3923 erwise it would not be possible to compute the witness W_4 , which is needed to solve the second
 3924 constraint. Witness W_4 therefore expresses the constraining of an intermediate computational
 3925 state.

3926 At this point, one might ask why equation 1 constrains the system to compute $W_1 \cdot W_2$ first,
 3927 since computing $W_2 \cdot W_3$, or $W_1 \cdot W_3$ in the beginning and then multiplying with the remaining
 3928 factor gives the exact same result. The reason is that the way we designed the R1CS prohibits
 3929 any of these alternative computations, which shows that R1CS are in general **not unique** de-
 3930 scriptions of a language: many different R1CS are able to describe the same problem.

To see that the two quadratic equations qualify as a rank-1 constraint system, choose the
 parameter $n = 1$, $m = 4$ and $k = 2$ as well as

$$\begin{array}{cccccc} a_0^1 = 0 & a_1^1 = 0 & a_2^1 = 1 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\ a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 0 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 1 \end{array}$$

$$\begin{array}{cccccc} b_0^1 = 0 & b_1^1 = 0 & b_2^1 = 0 & b_3^1 = 1 & b_4^1 = 0 & b_5^1 = 0 \\ b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 0 & b_3^2 = 0 & b_4^2 = 1 & b_5^2 = 0 \end{array}$$

$$\begin{array}{cccccc} c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 0 & c_4^1 = 0 & c_5^1 = 1 \\ c_0^2 = 0 & c_1^2 = 1 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 0 & c_5^2 = 0 \end{array}$$

With this choice, the rank-1 constraint system of our 3-factorization problem can be written in
 its most general form as follows:

$$\begin{aligned} (a_0^1 + a_1^1 I_1 + a_2^1 W_1 + a_3^1 W_2 + a_4^1 W_3 + a_5^1 W_4) \cdot (b_0^1 + b_1^1 I_1 + b_2^1 W_1 + b_3^1 W_2 + b_4^1 W_3 + b_5^1 W_4) &= (c_0^1 + c_1^1 I_1 + c_2^1 W_1 + c_3^1 W_2 + c_4^1 W_3 + c_5^1 W_4) \\ (a_0^2 + a_1^2 I_1 + a_2^2 W_2 + a_3^2 W_2 + a_4^2 W_3 + a_5^2 W_4) \cdot (b_0^2 + b_1^2 I_1 + b_2^2 W_2 + b_3^2 W_2 + b_4^2 W_3 + b_5^2 W_4) &= (c_0^2 + c_1^2 I_1 + c_2^2 W_2 + c_3^2 W_2 + c_4^2 W_3 + c_5^2 W_4) \end{aligned}$$

3931 *Example 112* (The Tiny Jubjub curve). Consider the languages $L_{\text{tiny.jj.1}}$ from example 107,
 3932 which consist of words (I_1, I_2) over the alphabet \mathbb{F}_{13} such that $3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2$.

check
reference

We derive a rank-1 constraint system such that its associated language is equivalent to
 $L_{\text{tiny.jj.1}}$. To achieve this, we first rewrite the defining equation:

$$\begin{aligned} 3 \cdot I_1^2 + I_2^2 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 && \Leftrightarrow \\ 0 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 - 3 \cdot I_1^2 - I_2^2 && \Leftrightarrow \\ 0 &= 1 + 8 \cdot I_1^2 \cdot I_2^2 + 10 \cdot I_1^2 + 12 \cdot I_2^2 \end{aligned}$$

Since R1CSs are systems of quadratic equations, we have to reformulate this expression into
 a system of quadratic equations. To do so, we have to introduce new variables that constrain
 intermediate steps in the computation and we have to decide if those variables should be public

or private. We decide to declare all new variables as private and get the following constraints

$$\begin{aligned}
 I_1 \cdot I_1 &= W_1 && \text{constraint 1} \\
 I_2 \cdot I_2 &= W_2 && \text{constraint 2} \\
 (8 \cdot W_1) \cdot W_2 &= W_3 && \text{constraint 3} \\
 (12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 &= 0 && \text{constraint 4}
 \end{aligned}$$

To see that these four quadratic equations qualify as a rank-1 constraint system according to definition XXX, choose the parameter $n = 2$, $m = 3$ and $k = 4$ as well as

add reference

$$\begin{aligned}
 a_0^1 &= 0 & a_1^1 &= 1 & a_2^1 &= 0 & a_3^1 &= 0 & a_4^1 &= 0 & a_5^1 &= 0 \\
 a_0^2 &= 0 & a_1^2 &= 0 & a_2^2 &= 1 & a_3^2 &= 0 & a_4^2 &= 0 & a_5^2 &= 0 \\
 a_0^3 &= 0 & a_1^3 &= 0 & a_2^3 &= 0 & a_3^3 &= 8 & a_4^3 &= 0 & a_5^3 &= 0 \\
 a_0^4 &= 1 & a_1^4 &= 0 & a_2^4 &= 0 & a_3^4 &= 10 & a_4^4 &= 12 & a_5^4 &= 1 \\
 \\
 b_0^1 &= 0 & b_1^1 &= 1 & b_2^1 &= 0 & b_3^1 &= 0 & b_4^1 &= 0 & b_5^1 &= 0 \\
 b_0^2 &= 0 & b_1^2 &= 0 & b_2^2 &= 1 & b_3^2 &= 0 & b_4^2 &= 0 & b_5^2 &= 0 \\
 b_0^3 &= 0 & b_1^3 &= 0 & b_2^3 &= 0 & b_3^3 &= 0 & b_4^3 &= 1 & b_5^3 &= 0 \\
 b_0^4 &= 1 & b_1^4 &= 0 & b_2^4 &= 0 & b_3^4 &= 0 & b_4^4 &= 0 & b_5^4 &= 0 \\
 \\
 c_0^1 &= 0 & c_1^1 &= 0 & c_2^1 &= 0 & c_3^1 &= 1 & c_4^1 &= 0 & c_5^1 &= 0 \\
 c_0^2 &= 0 & c_1^2 &= 0 & c_2^2 &= 0 & c_3^2 &= 0 & c_4^2 &= 1 & c_5^2 &= 0 \\
 c_0^3 &= 0 & c_1^3 &= 0 & c_2^3 &= 0 & c_3^3 &= 0 & c_4^3 &= 0 & c_5^3 &= 1 \\
 c_0^4 &= 0 & c_1^4 &= 0 & c_2^4 &= 0 & c_3^4 &= 0 & c_4^4 &= 0 & c_5^4 &= 0
 \end{aligned}$$

With this choice, the rank-1 constraint system of our tiny-jubjub curve point problem can be written in its most general form as follows:

$$\begin{aligned}
 (a_0^1 + a_1^1 I_1 + a_2^1 I_2 + a_3^1 W_1 + a_4^1 W_2 + a_5^1 W_3) \cdot (b_0^1 + b_1^1 I_1 + b_2^1 I_2 + b_3^1 W_1 + b_4^1 W_2 + b_5^1 W_3) &= (c_0^1 + c_1^1 I_1 + c_2^1 I_2 + c_3^1 W_1 + c_4^1 W_2 + c_5^1 W_3) \\
 (a_0^2 + a_1^2 I_1 + a_2^2 I_2 + a_3^2 W_1 + a_4^2 W_2 + a_5^2 W_3) \cdot (b_0^2 + b_1^2 I_1 + b_2^2 I_2 + b_3^2 W_1 + b_4^2 W_2 + b_5^2 W_3) &= (c_0^2 + c_1^2 I_1 + c_2^2 I_2 + c_3^2 W_1 + c_4^2 W_2 + c_5^2 W_3) \\
 (a_0^3 + a_1^3 I_1 + a_2^3 I_2 + a_3^3 W_1 + a_4^3 W_2 + a_5^3 W_3) \cdot (b_0^3 + b_1^3 I_1 + b_2^3 I_2 + b_3^3 W_1 + b_4^3 W_2 + b_5^3 W_3) &= (c_0^3 + c_1^3 I_1 + c_2^3 I_2 + c_3^3 W_1 + c_4^3 W_2 + c_5^3 W_3) \\
 (a_0^4 + a_1^4 I_1 + a_2^4 I_2 + a_3^4 W_1 + a_4^4 W_2 + a_5^4 W_3) \cdot (b_0^4 + b_1^4 I_1 + b_2^4 I_2 + b_3^4 W_1 + b_4^4 W_2 + b_5^4 W_3) &= (c_0^4 + c_1^4 I_1 + c_2^4 I_2 + c_3^4 W_1 + c_4^4 W_2 + c_5^4 W_3)
 \end{aligned}$$

3933 In what follows, we write L_{jubjub} for the associated language that consists of solutions to the
 3934 R1CS.

3935 To see that L_{jubjub} is equivalent to $L_{\text{tiny.jj.1}}$, let $(I_1, I_2; W_1, W_2, W_3)$ be a word in L_{jubjub} , then
 3936 (I_1, I_2) is a word in $L_{\text{tiny.jj.1}}$, since the defining R1CS of L_{jubjub} implies that I_1 and I_2 satisfy the
 3937 Edwards equation of the tiny jubjub curve. On the other hand, let (I_1, I_2) be a word in $L_{\text{tiny.jj.1}}$.
 3938 Then $(I_1, I_2; I_1^2, I_2^2, 8 \cdot I_1^2 \cdot I_2^2)$ is a word in L_{jubjub} and both maps are inverses of each other.

3939 **Exercise 43.** Consider the language $L_{\text{tiny.jj.zk}}$ and define a rank-1 constraint relation with a
 3940 decision function such that the associated language is equivalent to $L_{\text{tiny.jj.zk}}$.

3941 **R1CS Satisfiability** To understand how rank-1 constraint systems define formal languages,
 3942 observe that every R1CS over a field \mathbb{F} defines a decision function over the alphabet $\Sigma_I \times \Sigma_W =$
 3943 $\mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{\text{R1CS}} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{\text{true}, \text{false}\} ; (I; W) \mapsto \begin{cases} \text{true} & (I; W) \text{ satisfies R1CS} \\ \text{false} & \text{else} \end{cases} \quad (6.7)$$

Every R1CS therefore defines a formal language. The grammar of this language is encoded in the constraints, words are solutions to the equations and a **statement** is a knowledge claim “Given instance I , there is a witness W such that $(I; W)$ is a solution to the rank-1 constraint system”. A constructive proof to this claim is therefore an assignment of a field element to every witness variable, which is verified whenever the set of all instance and witness variables solves the R1CS.

Remark 2 (R1CS satisfiability). It should be noted that in our definition, every R1CS defines its own language. However, in more theoretical approaches, another language usually called **R1CS satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness or the computational complexity of the class of **all** R1CS. From our perspective, the R1CS satisfiability language is obtained by the union of all R1CS languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

$$L_{R1CS_SAT}(\mathbb{F}) = \{(i; w) \in \Sigma^* \times \Sigma^* \mid \text{there is a R1CS } R \text{ such that } R(i; w) = \text{true}\}$$

Example 113 (3-Factorization). Consider the language $L_{3.fac_zk}$ from example 106 and the R1CS defined in example ex:3-factorization-r1cs. As we have seen in ex:3-factorization-r1cs, solutions to the R1CS are in 1:1 correspondence with solutions to the decision function of $L_{3.fac_zk}$. Both languages are therefore equivalent in the sense that there is a 1:1 correspondence between words in both languages.

To give an intuition of what constructive proofs in $L_{3.fac_zk}$ look like, consider the instance $I_1 = 11$. To prove the statement “There exists a witness W such that $(I_1; W)$ is a word in $L_{3.fac_zk}$ ” constructively, a proof has to provide assignments to all witness variables W_1, W_2, W_3 and W_4 . Since the alphabet is \mathbb{F}_{13} , an example assignment is given by $W = (2, 3, 4, 6)$ since $(I_1; W)$ satisfies the R1CS

$$\begin{array}{ll} W_1 \cdot W_2 = W_4 & \# 2 \cdot 3 = 6 \\ W_4 \cdot W_3 = I_1 & \# 6 \cdot 4 = 11 \end{array}$$

A proper constructive proof is therefore given by $P = (2, 3, 4, 6)$. Of course, P is not the only possible proof for this statement. Since factorization is not unique in a field in general, another constructive proof is given by $P' = (3, 5, 12, 2)$.

Example 114 (The tiny jubjub curve). Consider the language L_{jubbub} from example 107 and its associated R1CS. To see how constructive proofs in L_{jubbub} look like, consider the instance $(I_1, I_2) = (11, 6)$. To prove the statement “There exists a witness W such that $(I_1, I_2; W)$ is a word in L_{jubbub} ” constructively, a proof has to provide assignments to all witness variables W_1, W_2 and W_3 . Since the alphabet is \mathbb{F}_{13} , an example assignment is given by $W = (4, 10, 8)$ since $(I_1, I_2; W)$ satisfies the R1CS

$$\begin{array}{ll} I_1 \cdot I_1 = W_1 & 11 \cdot 11 = 4 \\ I_2 \cdot I_2 = W_2 & 6 \cdot 6 = 10 \\ (8 \cdot W_1) \cdot W_2 = W_3 & (8 \cdot 4) \cdot 10 = 8 \\ (12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 = 0 & 12 \cdot 10 + 8 + 10 \cdot 4 + 1 = 0 \end{array}$$

A proper constructive proof is therefore given by $P = (4, 10, 8)$, which shows that the instance $(11, 6)$ is a point on the tiny jubjub curve.

check
referencecheck
referencecheck
referencecheck
reference

Modularity As we discussed on page 133 XXX, it is often useful to construct complex statements and their representing languages from simple ones. Rank-1 constraint systems are particularly useful for this, as the intersection of two R1CS over the same alphabet results in a new R1CS over that same alphabet.

check
reference

To be more precise, let S_1 and S_2 be two R1CS over \mathbb{F} , then a new R1CS S_3 is obtained by the intersection $S_3 = S_1 \cap S_2$ of S_1 and S_2 . In this context, intersection means that both the equations of S_1 **and** the equations of S_2 have to be satisfied in order to provide a solution for the system S_3 .

As a consequence, developers are able to construct complex R1CS from simple ones and this modularity provides the theoretical foundation for many R1CS compilers, as we will see in XXX.

add refer-
ence

6.2.2 Algebraic Circuits

As we have seen in the previous paragraphs, rank-1 constraint systems are quadratic equations such that solutions are knowledge proofs for the existence of words in associated languages. From the perspective of a proofer, it is therefore important to solve those equations efficiently.

However, in contrast to systems of linear equation, no general methods are known that solve systems of quadratic equations efficiently. Rank-1 constraint systems are therefore impractical from a proofers perspective and auxiliary information is needed that helps to compute solutions efficiently.

Methods which compute R1CS solutions are sometimes called **witness generator functions**. To provide a common example, we introduce another class of decision functions called **algebraic circuits**. As we will see, every algebraic circuit defines an associated R1CS and also provides an efficient way to compute solutions for that R1CS.

It can be shown that every space- and time-bounded computation is expressible as an algebraic circuit. Transforming high-level computer programs into those circuits is a process often called **flattening**.

To understand this in more detail, we will introduce our model for algebraic circuits and look at the concept of circuit execution and valid assignments. After that, we will show how to derive rank-1 constraint systems from circuits and how circuits are useful to compute solutions to their R1CS efficiently.

Algebraic circuit representation To see what algebraic circuits are, let \mathbb{F} be a field. An algebraic circuit is then a directed acyclic (multi)graph that computes a polynomial function over \mathbb{F} . Nodes with only outgoing edges (source nodes) represent the variables and constants of the function and nodes with only incoming edges (sink nodes) represent the outcome of the function. All other nodes have exactly two incoming edges and represent the defining field operations **addition** as well as **multiplication**. Graph edges represent the flow of the computation along the nodes.

To be more precise, we call a directed acyclic multi-graph $C(\mathbb{F})$ an **algebraic circuit** over \mathbb{F} in this book if the following conditions hold:

- The set of edges has a total order.
- Every source node has a label that represents either a variable or a constant from the field \mathbb{F} .
- Every sink node has exactly one incoming edge and a label that represents either a variable or a constant from the field \mathbb{F} .

- Every node that is neither a source nor a sink has exactly two incoming edges and a label from the set $\{+, *\}$ that represents either addition or multiplication in \mathbb{F} .
- All outgoing edges from a node have the same label.
- Outgoing edges from a node with a label that represents a variable have a label.
- Outgoing edges from a node with a label that represents multiplication have a label, if there is at least one labeled edge in both input path.
- All incoming edges to sink nodes have a label.
- If an edge has two labels S_i and S_j it gets a new label $S_i = S_j$.
- No other edge has a label.
- Incoming edges to sink nodes that are labeled with a constant $c \in \mathbb{F}$ are labeled with the same constant. Every other edge label is taken from the set $\{W, I\}$ and indexed compatible with the order of the edge set.

It should be noted that the details in the definitions of algebraic circuits vary between different sources. We use this definition as it is conceptually straightforward and well-suited for pen-and-paper computations.

To get a better intuition of our definition, let $C(\mathbb{F})$ be an algebraic circuit. Source nodes are the inputs to the circuit and either represent variables or constants. In a similar way, sink nodes represent termination points of the circuit and are either output variables or constants. Constant sink nodes enforce computational outputs to take on certain values.

Nodes that are neither source nodes nor sink nodes are called **arithmetic gates**. Arithmetic gates that are decorated with the “+”-label are called **addition-gates** and arithmetic gates that are decorated with the “·”-label are called **multiplication-gates**. Every arithmetic gate has exactly two inputs, represented by the two incoming edges.

Since the set of edges is ordered, we can write it as $\{E_1, E_2, \dots, E_n\}$ for some $n \in \mathbb{N}$ and we use those indices to index the edge labels, too. Edge labels are therefore either constants or symbols like I_j , W_j or S_j , where j is an index compatible with the edge order. Labels I_j represent instance variables, labels W_j witness variables. Labels on the outgoing edges of input variables constrain the associated variable to that edge. Every other edge defines a constraining equation in the associated R1CS. we will explain this in more detail in XXX.

Notation and Symbols 10. In synthesizing algebraic circuits, assigning instance I_j or witness W_j labels to appropriate edges is often the final step. It is therefore convenient to not distinguish these two types of edges in previous steps. To account for that, we often simply write S_j for an edge label, indicating that the private/public property of the label is unspecified and it might represent an instance or a witness label.

Example 115 (Generalized factorization SNARK). To give a simple example of an algebraic circuit, consider our 3-factorization problem from example 106 again. To express the problem in the algebraic circuit model, consider the following function

$$f_{3, fac} : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \rightarrow \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 \cdot x_3$$

Using this function, we can describe the zero-knowledge 3-factorization problem from 106, in the following way: Given instance $I_1 \in \mathbb{F}_{13}$, a valid witness is a preimage of $f_{3, fac}$ at

add reference

check reference

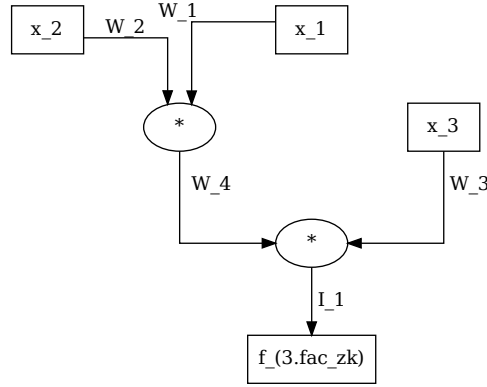
check reference

the point I_1 , i.e., a valid witness consists of three values W_1 , W_2 and W_3 from \mathbb{F}_{13} such that $f_{3.fac}(W_1, W_2, W_3) = I_1$.

To see how this function can be transformed into an algebraic circuit over \mathbb{F}_{13} , it is a common first step to introduce brackets into the function's definition and then write the operations as binary operators, in order to highlight how exactly every field operation acts on its two inputs. Due to the associativity laws in a field, we have several choices. We choose

$$\begin{aligned} f_{3.fac}(x_1, x_2, x_3) &= x_1 \cdot x_2 \cdot x_3 && \# \text{ bracket choice} \\ &= (x_1 \cdot x_2) \cdot x_3 && \# \text{ operator notation} \\ &= MUL(MUL(x_1, x_2), x_3) \end{aligned}$$

Using this expression, we can write an associated algebraic circuit by first constraining the variables to edge labels $W_1 = x_1$, $W_2 = x_2$ and $W_3 = x_3$ as well as $I_1 = f_{3.fac}(x_1, x_2, x_3)$, taking the distinction between private and public inputs into account. We then rewrite the operator representation of $f_{3.fac}$ into circuit nodes and get the following:



4049

In this case, the directed acyclic multi-graph is a binary tree with three leaves (the source nodes) labeled by x_1 , x_2 and x_3 , one root (the single sink node) labeled by $f(x_1, x_2, x_3)$ and two internal nodes, which are labeled as multiplication gates.

The order we used to label the edges is chosen to make the edge labeling consistent with the choice of W_4 as defined in example XXX. This order can be obtained by a depth-first right-to-left-first traversal algorithm.

add reference

Example 116. To give a more realistic example of an algebraic circuit, look at the defining equation XXX of the tiny-jubjub curve again. A pair of field elements $(x, y) \in \mathbb{F}_{13}^2$ is a curve point, precisely if

add reference

$$3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$$

To understand how one might transform this identity into an algebraic circuit, we first rewrite this equation by shifting all terms to the right. We get:

$$3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2 \quad \Leftrightarrow$$

$$0 = 1 + 8 \cdot x^2 \cdot y^2 - 3 \cdot x^2 - y^2 \quad \Leftrightarrow$$

$$0 = 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2$$

Then we use this expression to define a function such that all points of the tiny-jubjub curve are characterized as the function preimages at 0.

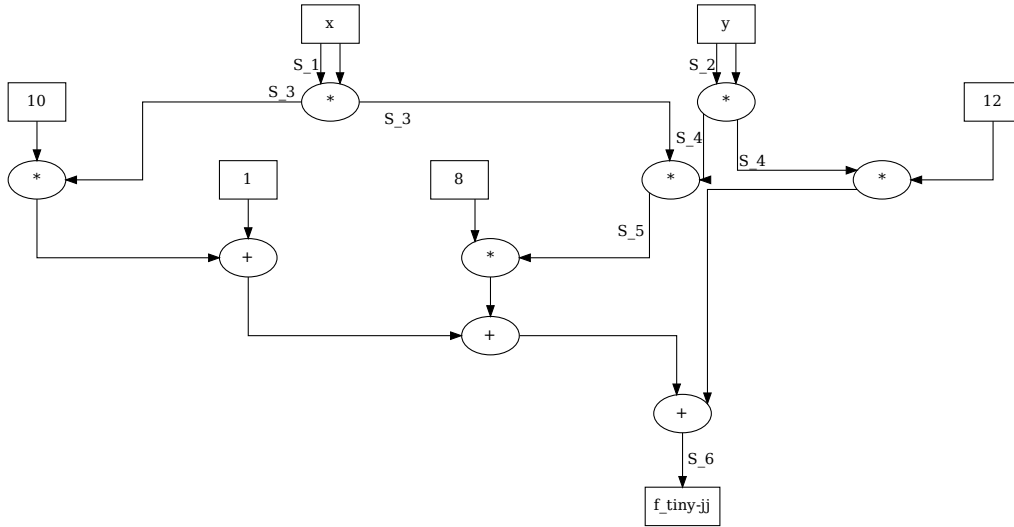
$$f_{\text{tiny-jj}} : \mathbb{F}_{13} \times \mathbb{F}_{13} \rightarrow \mathbb{F}_{13} ; (x, y) \mapsto 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2$$

Every pair of points $(x, y) \in \mathbb{F}_{13}^2$ with $f_{\text{tiny-jj}}(x, y) = 0$ is a point on the tiny-jubjub curve, and there are no other curve points. The preimage $f_{\text{tiny-jj}}^{-1}(0)$ is therefore a complete description of the tiny-jubjub curve.

We can transform this function into an algebraic circuit over \mathbb{F}_{13} . We first introduce brackets into potentially ambiguous expressions and then rewrite the function in terms of binary operators. We get

$$\begin{aligned} f_{\text{tiny-jj}}(x, y) &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12y^2 && \Leftrightarrow \\ &= ((8 \cdot ((x \cdot x) \cdot (y \cdot y))) + (1 + 10 \cdot (x \cdot x))) + (12 \cdot (y \cdot y)) && \Leftrightarrow \\ &= \text{ADD}(\text{ADD}(\text{MUL}(8, \text{MUL}(\text{MUL}(x, x), \text{MUL}(y, y))), \text{ADD}(1, \text{MUL}(10, \text{MUL}(x, x)))), \text{MUL}(12, \text{MUL}(y, y))) \end{aligned}$$

Since we haven't decided which part of the computation should be public and which part should be private, we use the unspecified symbol S to represent edge labels. Constraining all variables to edge labels $S_1 = x$, $S_2 = y$ and $S_6 = f_{\text{tiny-jj}}$, we get the following circuit, representing the function $f_{\text{tiny-jj}}$, by inductively replacing binary operators with their associated arithmetic gates:



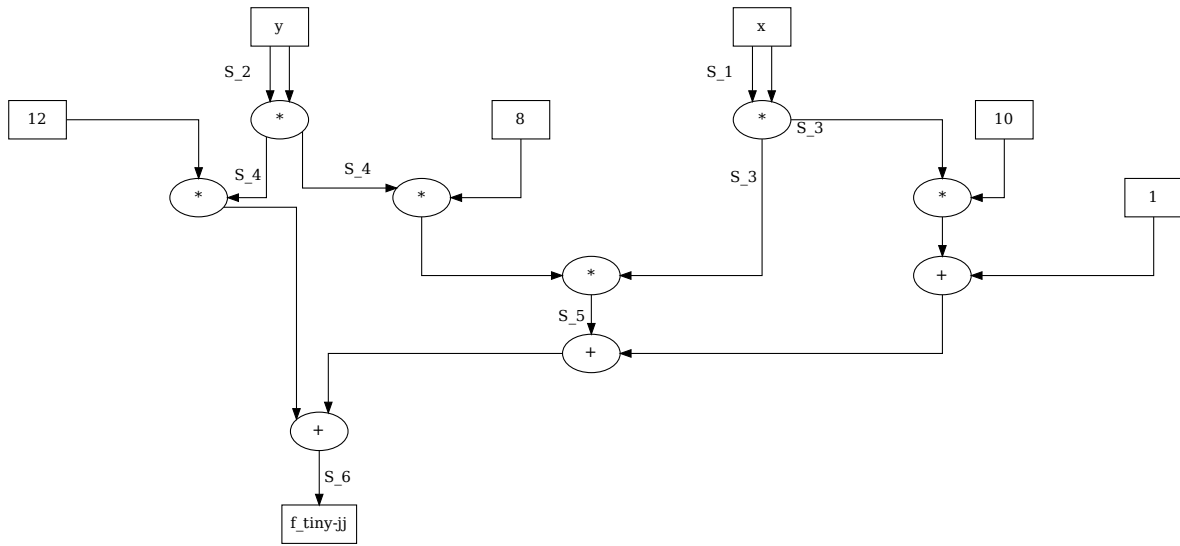
4064

This circuit is not a graph, but a multigraph, since there is more than one edge between some of the nodes.

In the process of designing of circuits from functions, it should be noted that circuit representations are not unique in general. In case of the function $f_{\text{tiny-jj}}$, the circuit shape is dependent on our choice of bracketing in XXX. An alternative design is, for example, given by the following circuit, which occurs when the bracketed expression $8 \cdot ((x \cdot x) \cdot (y \cdot y))$ is replaced by the expression $(x \cdot x) \cdot (8 \cdot (y \cdot y))$.

4072

 add refer-
ence



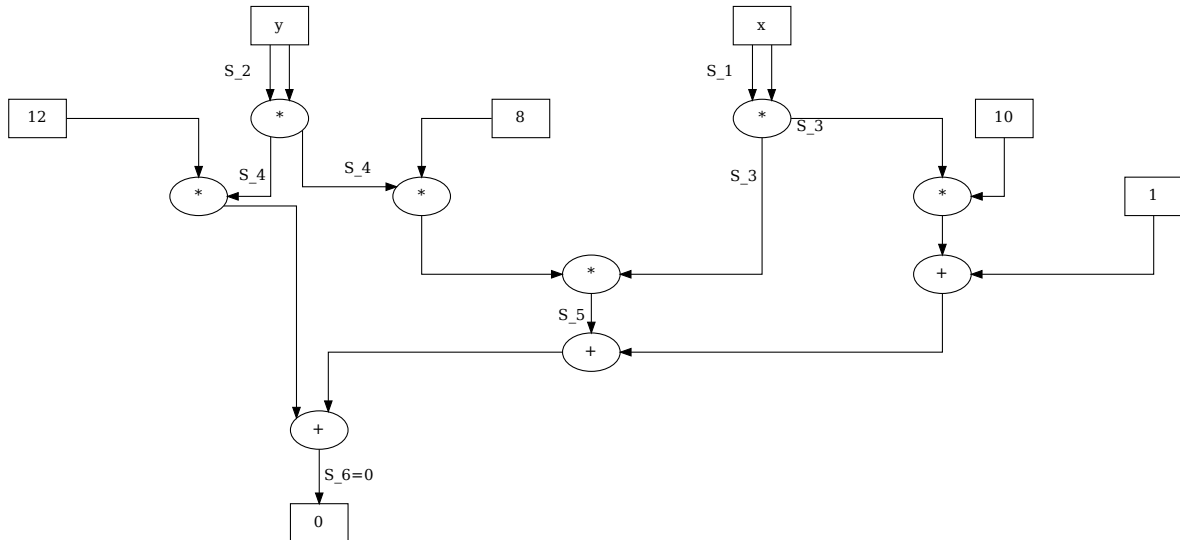
4073

4074

4075 Of course, both circuits represent the same function, due to the associativity and commutativity
 4076 laws that hold true in any field.

4077 With a circuit that represents the function $f_{\text{tiny-jj}}$, we can now proceed to derive a circuit
 4078 that constrains arbitrary pairs (x, y) of field elements to be points on the tiny-jubjub curve. To do
 4079 so, we have to constrain the output to be zero, that is, we have to constrain $S_6 = 0$. To indicate
 4080 this in the circuit, we replace the output variable by the constant 0 and constrain the related edge
 4081 label accordingly. We get

4082

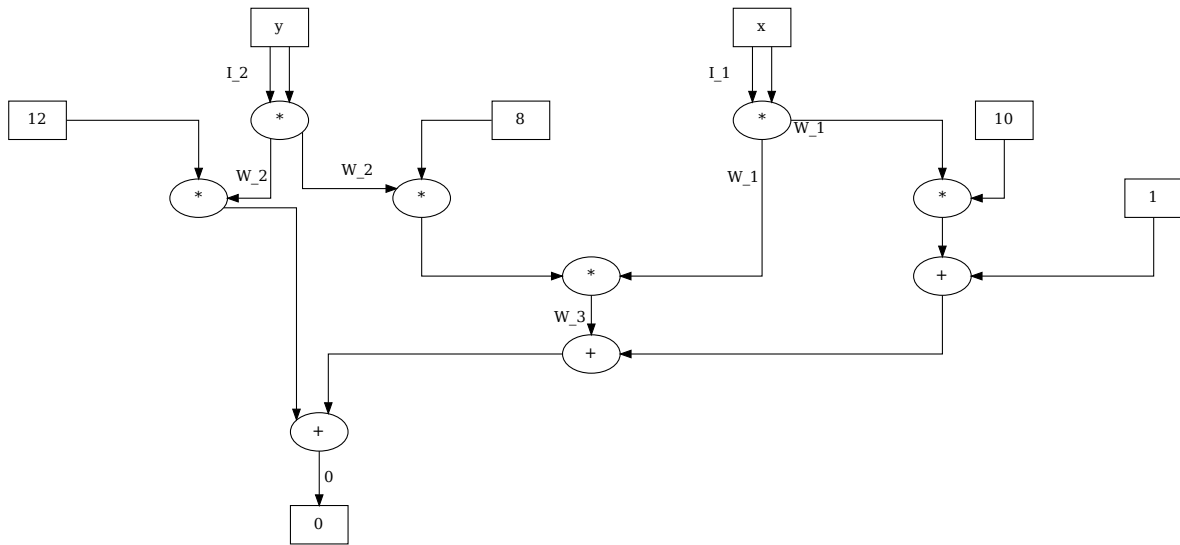


4083

4084

4085 The previous circuit enforces input values assigned to the labels S_1 and S_2 to be points on the
 4086 tiny jubjub curve. However, it does not specify which labels are considered public and which
 4087 are considered private. The following circuit defines the inputs to be public, while all other
 4088 labels are private:

4089



4090

4091

4092 It can be shown that every space- and time-bounded computation can be transformed into
 4093 an algebraic circuit. We call any process that transforms a bounded computation into a circuit
 4094 **flattening**.

We al-
ready said
this in
this chap-
ter

4095 **Circuit Execution** Algebraic circuits are directed, acyclic multi-graphs, where nodes repre-
 4096 sent variables, constants, or addition and multiplication gates. In particular, every algebraic
 4097 circuit with n input nodes decorated with variable symbols and m output nodes decorated with
 4098 variables can be seen a function that transforms an input tuple (x_1, \dots, x_n) from \mathbb{F}^n into an out-
 4099 put tuple (f_1, \dots, f_m) from \mathbb{F}^m . The transformation is done by sending values associated to
 4100 nodes along their outgoing edges to other nodes. If those nodes are gates, then the values are
 4101 transformed according to the gate label and the process is repeated along all edges until a sink
 4102 node is reached. We call this computation **circuit execution**.

4103 When executing a circuit, it is possible to not only compute the output values of the circuit
 4104 but to derive field elements for all edges, and, in particular, for all edge labels in the circuit. The
 4105 result is a tuple (S_1, S_2, \dots, S_n) of field elements associated to all labeled edges, which we call a
 4106 **valid assignment** to the circuit. In contrast, any assignment $(S'_1, S'_2, \dots, S'_n)$ of field elements to
 4107 edge labels that can not arise from circuit execution is called an **invalid assignment**.

4108 Valid assignments can be interpreted as **proofs for proper circuit execution** because they
 4109 keep a record of the computational result as well as intermediate computational steps.

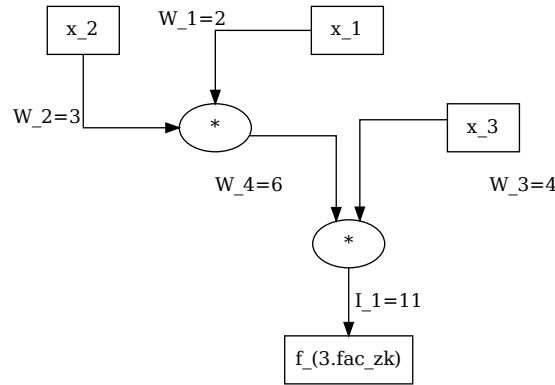
4110 *Example 117 (3-factorization).* Consider the 3-factorization problem from example 106 and its
 4111 representation as an algebraic circuit from XXX. We know that the set of edge labels is given
 4112 by $S := \{I_1; W_1, W_2, W_3, W_4\}$.

check
reference

4113 To understand how this circuit is executed, consider the variables $x_1 = 2$, $x_2 = 3$ as well as
 4114 $x_3 = 4$. Following all edges in the graph, we get the assignments $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$.
 4115 Then the assignments of W_1 and W_2 enter a multiplication gate and the output of the gate is
 4116 $2 \cdot 3 = 6$, which we assign to W_4 , i.e. $W_4 = 6$. The values W_4 and W_3 then enter the second
 4117 multiplication gate and the output of the gate is $6 \cdot 4 = 11$, which we assign to I_1 , i.e. $I_1 = 11$.

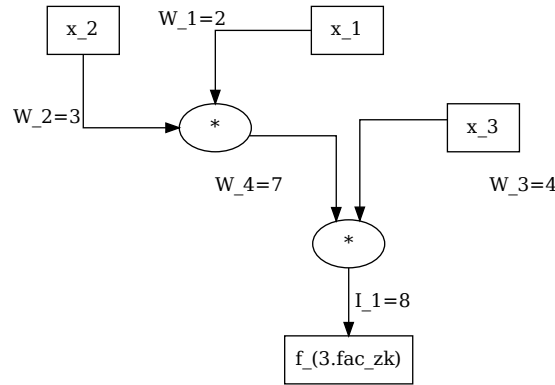
add refer-
ence

4118 A valid assignment to the 3-factorization circuit $C_{3, \text{fac}}(\mathbb{F}_{13})$ is therefore given by the set
 4119 $S_{\text{valid}} := \{11; 2, 3, 4, 6\}$. We can picture this assignment in the circuit as follows:



4120

4121 To see what an invalid assignment looks like, consider the assignment $S_{err} := \{8; 2, 3, 4, 7\}$. In
 4122 this assignment, the input values are the same as in the previous case. The associated circuit is:



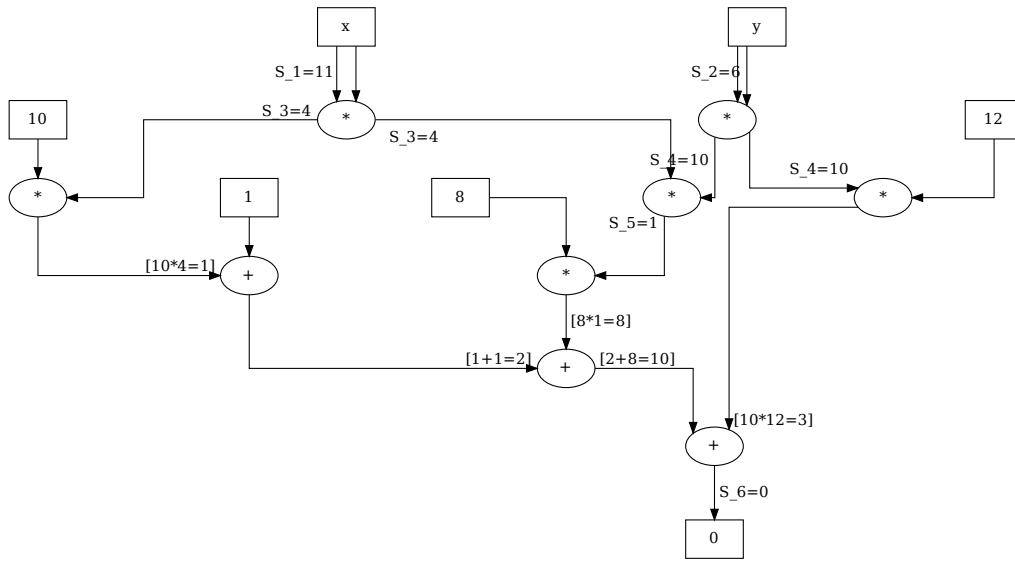
4123

4124 This assignment is invalid, as the assignments of I_1 and W_4 cannot be obtained by executing the
 4125 circuit.

4126 *Example 118.* To compute a more realistic algebraic circuit execution, consider the defining
 4127 circuit $C_{tiny-jj}(\mathbb{F}_{13})$ from example 114 again. We already know from the way this circuit is
 4128 constructed that any valid assignment with $S_1 = x$, $S_2 = y$ and $S_6 = 0$ will ensure that the pair
 4129 (x, y) is a point on the tiny jubjub curve XXX in its Edwards representation.

4130 From example 114 we know that the pair $(11, 6)$ is a proper point on the tiny-jubjub curve
 4131 and we use this point as input to a circuit execution. We get:

check
referenceadd refer-
encecheck
reference



4132

Executing the circuit, we indeed compute $S_6 = 0$ as expected, which proves that $(11, 6)$ is a point on the tiny-jubjub curve in its Edwards representation. A valid assignment of $C_{\text{tiny-jj}}(\mathbb{F}_{13})$ is therefore given by

$$S_{\text{tiny-jj}} = \{S_1, S_2, S_3, S_4, S_5, S_6\} = \{11, 6, 4, 10, 1, 0\}$$

4133 **Circuit Satisfiability** To understand how algebraic circuits give rise to formal languages, ob-
 4134 serve that every algebraic circuit $C(\mathbb{F})$ over a fields \mathbb{F} defines a decision function over the al-
 4135 phabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{C(\mathbb{F})} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{true, false\} ; (I; W) \mapsto \begin{cases} true & (I; W) \text{ is valid assignment to } C(\mathbb{F}) \\ false & \text{else} \end{cases} \quad (6.8)$$

4136 Every algebraic circuit therefore defines a formal language. The grammar of this language is
 4137 encoded in the shape of the circuit, words are assignments to edge labels that are derived from
 4138 circuit execution, and **statements** are knowledge claims “Given instance I , there is a witness
 4139 W such that $(I; W)$ is a valid assignment to the circuit”. A constructive proof to this claim
 4140 is therefore an assignment of a field element to every witness variable, which is verified by
 4141 executing the circuit to see if the assignment of the execution meets the assignment of the
 4142 proof.

4143 In the context of zero-knowledge proof systems, executing circuits is also often called **wit-**
 4144 **ness generation**, since in applications the instance part is usually public, while its the task of a
 4145 prover to compute the witness part.

Remark 3 (Circuit satisfiability). It should be noted that, in our definition, every circuit defines its own language. However, in more theoretical approaches another language usually called **circuit satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness, or computational complexity of the class of **all** algebraic circuits over a given field. From our perspective the circuit satisfiability language is obtained by union of all circuit languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

Should we refer to RICS satisfiability (p. 137 here?)

$$L_{\text{CIRCUIT_SAT}(\mathbb{F})} = \{(i; w) \in \Sigma^* \times \Sigma^* \mid \text{there is a circuit } C(\mathbb{F}) \text{ such that } (i; w) \text{ is valid assignment}\}$$

4146 *Example 119 (3-Factorization).* Consider the circuit $C_{3.fac}$ from example XXX again. We call
 4147 the associated language $L_{3.fac_circ}$.

add refer-
ence

4148 To understand how a constructive proof of a statement in $L_{3.fac_circ}$ looks like, consider the
 4149 instance $I_1 = 11$. To provide a proof for the statement “There exist a witness W such that $(I_1; W)$
 4150 is a word in $L_{3.fac_circ}$ ” a proof therefore has to consists of proper values for the variables W_1 ,
 4151 W_2 , W_3 and W_4 . Any proofer therefore has to find input values for W_1 , W_2 and W_3 and then
 4152 execute the circuit to compute W_4 under the assumption $I_1 = 11$.

4153 Example XXX implies that $(2, 3, 4, 6)$ is a proper constructive proof and in order to verify
 4154 the proof a verifier needs to execute the circuit with instance $I_1 = 11$ and inputs $W_1 = 2$, $W_2 = 3$
 4155 and $W_3 = 4$ to decide whether the proof is a valid assignment or not.

add refer-
ence

4156 **Associated Constraint Systems** As we have seen in XXX, rank-1 constraint systems define
 4157 a way to represent statements in terms of a system of quadratic equations over finite fields,
 4158 suitable for pairing-based zero-knowledge proof systems. However, those equations provide no
 4159 practical way for a proofer to actually compute a solution. On the other hand, algebraic circuits
 4160 can be executed in order to derive valid assignments efficiently.

add refer-
ence

4161 In this paragraph, we show how to transform any algebraic circuit into a rank-1 constraint
 4162 system such that valid circuit assignments are in 1:1 correspondence with solutions to the asso-
 4163 ciated R1CS.

4164 To see this, let $C(\mathbb{F})$ be an algebraic circuit over a finite field \mathbb{F} , with a set of edge labels
 4165 $\{S_1, S_2, \dots, S_n\}$. Then one of the following steps is executed for every edge label S_j from that
 4166 set:

- 4167 • If the edge label S_j is an outgoing edge of a multiplication gate, the R1CS gets a new
 4168 quadratic constraint

$$(\text{left input}) \cdot (\text{right input}) = S_j \quad (6.9)$$

4169 where (left input) respectively (right input) is the output from the symbolic execution of
 4170 the subgraph that consists of the left respectively right input edge of this gate, and all
 4171 edges and nodes that have this edge in their path, starting with constant inputs or labeled
 4172 outgoing edges of other nodes.

- 4173 • If the edge label S_j is an outgoing edge of an addition gate, the R1CS gets a new quadratic
 4174 constraint

$$(\text{left input} + \text{right input}) \cdot 1 = S_j \quad (6.10)$$

4175 where (left input) respectively (right input) is the output from the symbolic execution of
 4176 the subgraph that consists of the left respectively right input edge of this gate and all
 4177 edges and nodes that have this edge in their path, starting with constant inputs or labeled
 4178 outgoing edges of other nodes.

- 4179 • No other edge label adds a constraint to the system.

4180 The result of this method is a rank-1 constraint system, and in this sense, every algebraic circuit
 4181 $C(\mathbb{F})$ generates a R1CS R , which we call the **associated R1CS** of the circuit. It can be shown
 4182 that a tuple of field elements (S_1, S_2, \dots, S_n) is a valid assignment to a circuit if and only if the
 4183 same tuple is a solution to the associated R1CS. Circuit executions therefore compute solutions
 4184 to rank-1 constraints systems efficiently.

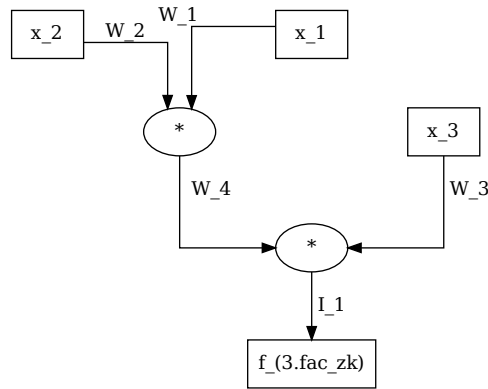
4185 To understand the contribution of algebraic gates to the number of constraints, note that by
 4186 definition multiplication gates have labels on their outgoing edges if and only if there is at least
 4187 one labeled edge in both input paths, or if the outgoing edge is an input to a sink node. This

implies that multiplication with a constant is essentially free in the sense that it doesn't add a new constraint to the system, as long as that multiplication gate is not an input to an output node.

Moreover, addition gates have labels on their outgoing edges if and only if they are inputs to sink nodes. This implies that addition is essentially free in the sense that it doesn't add a new constraint to the system, as long as that addition gate is not an input to an output node.

Example 120 (3-factorization). Consider our 3-factorization problem from example XXX and the associated circuit $C_{3, \text{fac}}(\mathbb{F}_{13})$. Our task is to transform this circuit into an equivalent rank-1 constraint system.

add reference



We start with an empty R1CS, and, in order to generate all constraints, we have to iterate over the set of edge labels $\{I_1; W_1, W_2, W_3, W_4\}$.

Starting with the edge label I_1 , we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$\begin{aligned} (\text{left input}) \cdot (\text{right input}) &= I_1 \\ W_4 \cdot W_3 &= I_1 \end{aligned} \quad \Leftrightarrow$$

Next, we consider the edge label W_1 and, since, it's not an outgoing edge of a multiplication or addition label, we don't add a constraint to the system. The same holds true for the labels W_2 and W_3 .

For edge label W_4 , we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$\begin{aligned} (\text{left input}) \cdot (\text{right input}) &= W_4 \\ W_2 \cdot W_1 &= W_4 \end{aligned} \quad \Leftrightarrow$$

Since there are no more labeled edges, all constraints are generated, and we have to combine them to get the associated R1CS of $C_{3, \text{fac}}(\mathbb{F}_{13})$:

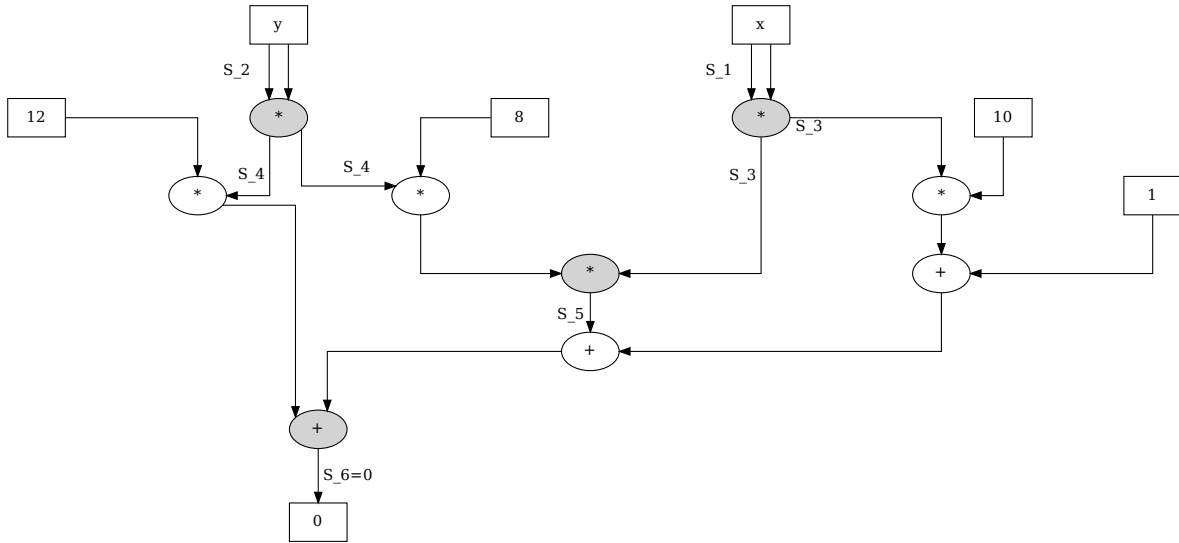
$$\begin{aligned} W_4 \cdot W_3 &= I_1 \\ W_2 \cdot W_1 &= W_4 \end{aligned}$$

This system is equivalent to the R1CS we derived in example 111. The languages $L_{3, \text{fac_zk}}$ and $L_{3, \text{fac_circ}}$ are therefore equivalent and both the circuit as well as the R1CS are just two different ways of expressing the same language.

check reference

4206 *Example 121.* To consider a more general transformation, we consider the tiny-jubjub circuit
 4207 from example 114 again. A proper circuit is given by

check
reference



4209

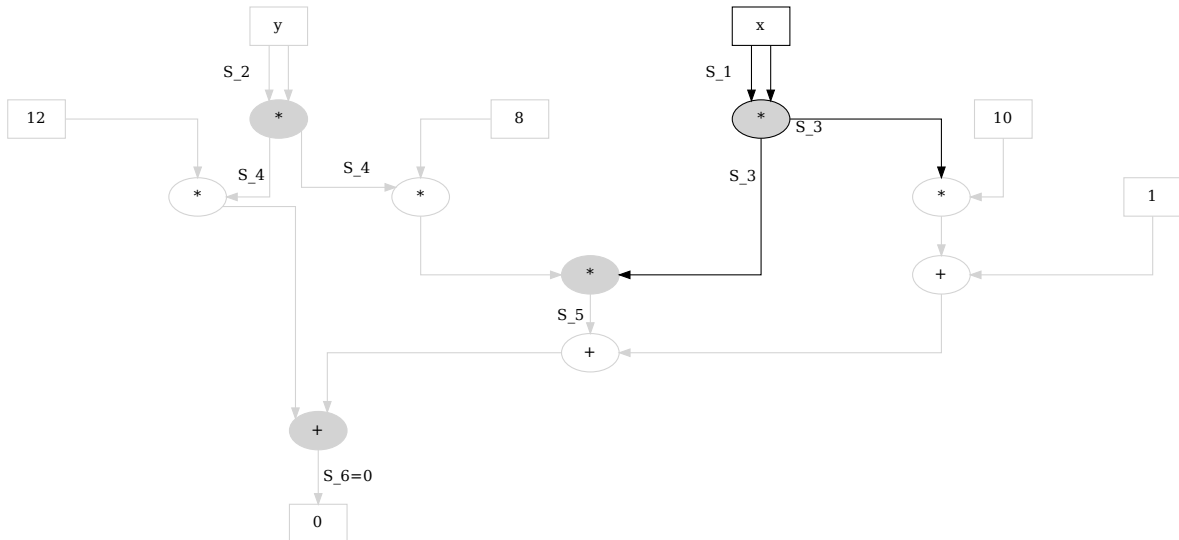
4210

4211 To compute the number of constraints, observe that we have 3 multiplication gates that have
 4212 labels on their outgoing edges and 1 addition gate that has a label on its outgoing edge. We
 4213 therefore have to compute 4 quadratic constraints.

4214 In order to derive the associated R1CS, we have start with an empty R1CS and then iterate
 4215 over the set $\{S_1, S_2, S_3, S_4, S_5, S_6 = 0\}$ of all edge labels, in order to generate the constraints.

4216 Considering edge label S_1 , we see that the associated edges are not outgoing edges of any
 4217 algebraic gate, and we therefore have to add no new constraint to the system. The same holds
 4218 true for edge label S_2 . Looking at edge label S_3 , we see that the associated edges are outgoing
 4219 edges of a multiplication gate and that the associated subgraph is given by:

4220



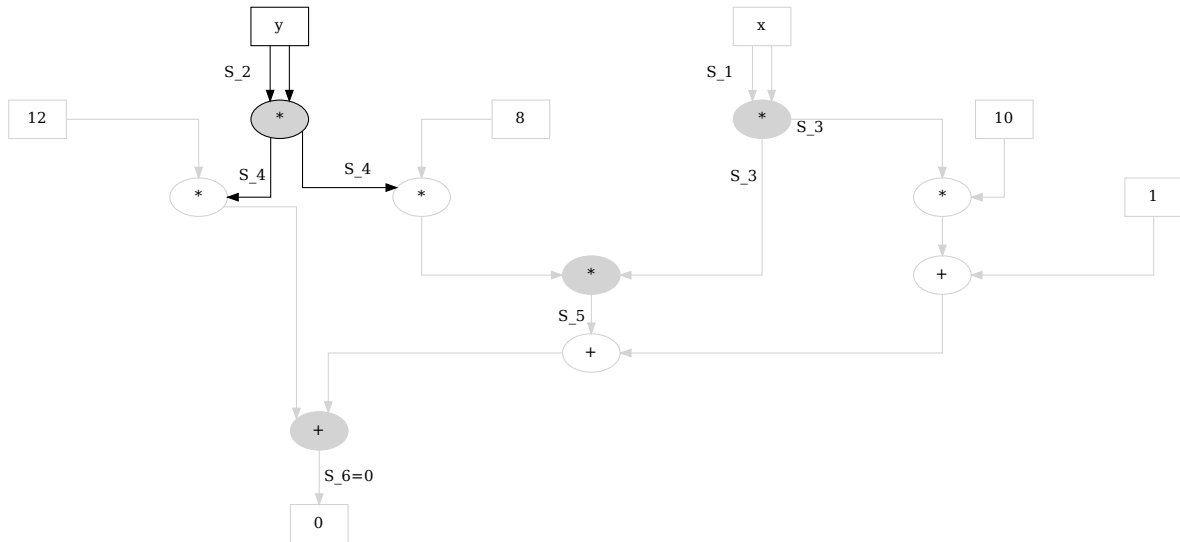
4221

4222

Both the left and the right input to this multiplication gate are labeled by S_1 . We therefore have to add the following constraint to the system:

$$S_1 \cdot S_1 = S_3$$

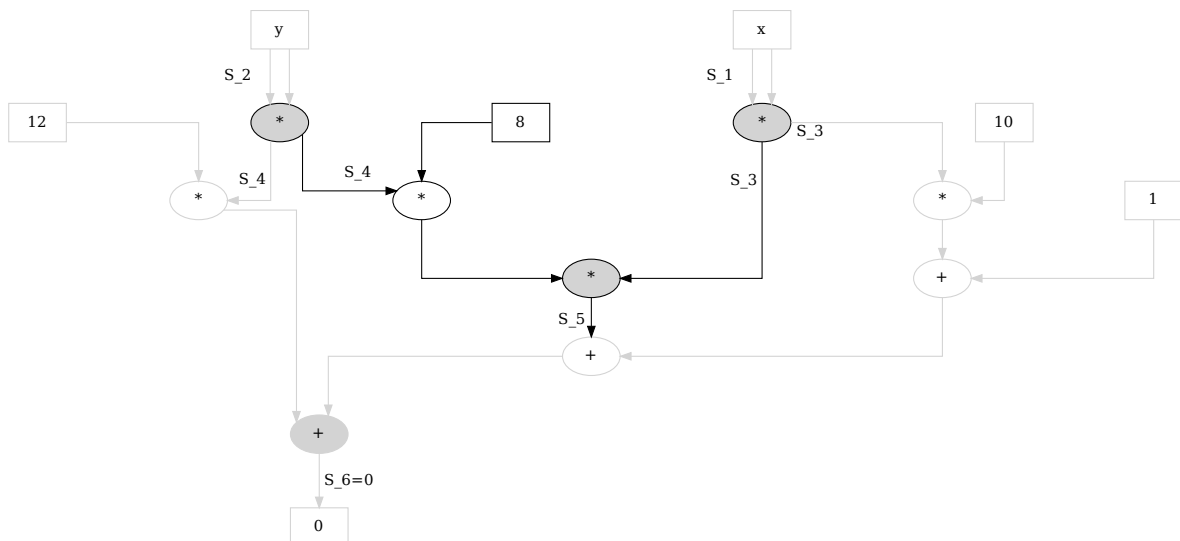
Looking at edge label S_4 , we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:



Both the left and the right input to this multiplication gate are labeled by S_2 and we therefore have to add the following constraint to the system:

$$S_2 \cdot S_2 = S_4$$

Edge label S_5 is more interesting. To see if it implies a constraint, we have to construct the associated subgraph first, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get



4233

The right input to the associated multiplication gate is given by the labeled edge S_3 . However, the left input is not a labeled edge, but has a labeled edge in one of its path. This implies that we have to add a constraint to the system. To compute the left factor of that constraint, we have to compute the output of subgraph associated to the left edge, which is $8 \cdot W_2$. This gives the constraint

$$(S_4 \cdot 8) \cdot S_3 = S_5$$

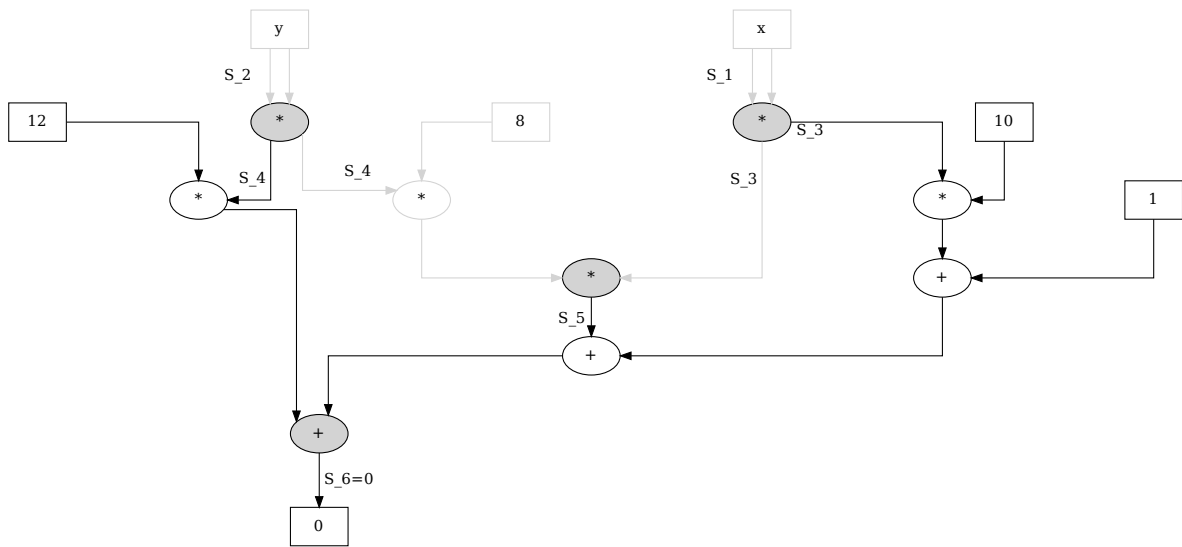
4234

4235

4236

The last edge label is the constant $S_6 = 0$. To see if it implies a constraint, we have to construct the associated subgraph, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get

4237



4238

4239

Both the left and the right input are unlabeled, but have a labeled edges in their path. This implies that we have to add a constraint to the system. Since the gate is an addition gate, the right factor in the quadratic constraint is always 1 and the left factor is computed by symbolically executing all inputs to all gates in sub-circuit. We get

$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

Since there are no more labeled outgoing edges, we are done deriving the constraints. Combining all constraints together, we get the following R1CS:

$$S_1 \cdot S_1 = S_3$$

$$S_2 \cdot S_2 = S_4$$

$$(S_4 \cdot 8) \cdot S_3 = S_5$$

$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

4240

4241

4242

which is equivalent to the R1CS we derived in example 114. The languages L_{3, fac_zk} and L_{3, fac_circ} are therefore equivalent and both the circuit as well as the R1CS are just two different ways to express the same language.

 check
reference

6.2.3 Quadratic Arithmetic Programs

We have introduced algebraic circuits and their associated rank-1 constraints systems as two particular models able to represent space- and time-bounded computation. Both models define formal languages, and associated membership as well as knowledge claims can be constructively proved by executing the circuit in order to compute solutions to its associated RICS.

One reason why those systems are useful in the context of succinct zero-knowledge proof systems is because any RICS can be transformed into another computational model called **quadratic arithmetic programs** (QAP), which serve as the basis for some of the most efficient succinct non-interactive zero-knowledge proof generators that currently exist.

As we will see, proving statements for languages that have checking relations defined by quadratic arithmetic programs can be achieved by providing certain polynomials, and those proofs can be verified by checking a particular divisibility property.

QAP representation To understand what quadratic arithmetic programs are in detail, let \mathbb{F} be a field and R a rank-1 constraints system over \mathbb{F} such that the number of non-zero elements in \mathbb{F} is strictly larger than the number k of constraints in R . Moreover, let a_j^i, b_j^i and $c_j^i \in \mathbb{F}$ for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$, be the defining constants of the RICS and m_1, \dots, m_k be arbitrary, invertible and distinct elements from \mathbb{F} .

Then a **quadratic arithmetic program** [QAP] of the RICS is the following set of polynomials over \mathbb{F}

$$QAP(R) = \left\{ T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{h=0}^{n+m} \right\} \quad (6.11)$$

where $T(x) := \prod_{i=1}^k (x - m_i)$ is a polynomial of degree k , called the **target polynomial** of the QAP and A_j, B_j as well as C_j are the unique degree $k-1$ polynomials defined by the equations

$$A_j(m_i) = a_j^i \quad B_j(m_i) = b_j^i \quad C_j(m_i) = c_j^i \quad j = 1, \dots, n+m+1, i = 1, \dots, k \quad (6.12)$$

Given some rank-1 constraint system, an associated quadratic arithmetic program is therefore nothing but a set of polynomials, computed from the constants in the RICS. To see that the polynomials A_j, B_j and C_j are uniquely defined by the equations in XXX, recall that a polynomial of degree $k-1$ is completely determined on k evaluation points and the equation XXX precisely determines those k evaluation points.

Since we only consider polynomials over fields, Lagrange's interpolation method from 3.31 in chapter 3 can be used to derive the polynomials A_j, B_j and C_j from their defining equations XXX. A practical method to compute a QAP from a given RICS therefore consists of two steps. If the RICS consists of k constraints, first choose k invertible and mutually different points from the underlying field. Every choice defines a different QAP for the same RICS. Then use Lagrange's method and equation XXX to compute the polynomials A_j, B_j and C_j for every $1 \leq j \leq k$.

Example 122 (Generalized factorization SNARK). To provide a better intuition of quadratic arithmetic programs and how they are computed from their associated rank-1 constraint systems, consider the language $L_{3.fac_zk}$ from example 106 and its associated RICS

$$\begin{array}{ll} W_1 \cdot W_2 = W_4 & \text{constraint 1} \\ W_4 \cdot W_3 = I_1 & \text{constraint 2} \end{array}$$

In this example we want to transform this RICS into an associated QAP. In a first step, we have to compute the defining constants a_j^i, b_j^i and c_j^i of the RICS. According to XXX, we have

add reference

"by"?

add reference

check reference

add reference

add reference

check reference

add reference

$$\begin{array}{cccccc} a_0^1 = 0 & a_1^1 = 0 & a_2^1 = 1 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\ a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 0 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 1 \end{array}$$

$$\begin{array}{cccccc} b_0^1 = 0 & b_1^1 = 0 & b_2^1 = 0 & b_3^1 = 1 & b_4^1 = 0 & b_5^1 = 0 \\ b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 0 & b_3^2 = 0 & b_4^2 = 1 & b_5^2 = 0 \end{array}$$

$$\begin{array}{cccccc} c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 0 & c_4^1 = 0 & c_5^1 = 1 \\ c_0^2 = 0 & c_1^2 = 1 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 0 & c_5^2 = 0 \end{array}$$

Since the R1CS is defined over the field \mathbb{F}_{13} and has two constraining equations, we need to choose two arbitrary but distinct elements m_1 and m_2 from \mathbb{F}_{13} . We choose $m_1 = 5$, and $m_2 = 7$ and with this choice we get the target polynomial

$$\begin{aligned} T(x) &= (x - m_1)(x - m_2) && \# \text{ Definition of } T \\ &= (x - 5)(x - 7) && \# \text{ Insert our choice} \\ &= (x + 8)(x + 6) && \# \text{ Negatives in } \mathbb{F}_{13} \\ &= x^2 + x + 9 && \# \text{ expand} \end{aligned}$$

4276 Then we have to compute the polynomials A_j , B_j and C_j by their defining equation from the
4277 R1CS coefficients. Since the R1CS has two constraining equations, those polynomials are of
4278 degree 1 and they are defined by their evaluation at the point $m_1 = 5$ and the point $m_2 = 7$.

At point m_1 , each polynomial A_j is defined to be a_j^1 and at point m_2 , each polynomial A_j is defined to be a_j^2 . The same holds true for the polynomials B_j as well as C_j . Writing all these equations now, we get:

$$\begin{array}{l} A_0(5) = 0, \quad A_1(5) = 0, \quad A_2(5) = 1, \quad A_3(5) = 0, \quad A_4(5) = 0, \quad A_5(5) = 0 \\ A_0(7) = 0, \quad A_1(7) = 0, \quad A_2(7) = 0, \quad A_3(7) = 0, \quad A_4(7) = 0, \quad A_5(7) = 1 \end{array}$$

$$\begin{array}{l} B_0(5) = 0, \quad B_1(5) = 0, \quad B_2(5) = 0, \quad B_3(5) = 1, \quad B_4(5) = 0, \quad B_5(5) = 0 \\ B_0(7) = 0, \quad B_1(7) = 0, \quad B_2(7) = 0, \quad B_3(7) = 0, \quad B_4(7) = 1, \quad B_5(7) = 0 \end{array}$$

$$\begin{array}{l} C_0(5) = 0, \quad C_1(5) = 0, \quad C_2(5) = 0, \quad C_3(5) = 0, \quad C_4(5) = 0, \quad C_5(5) = 1 \\ C_0(7) = 0, \quad C_1(7) = 1, \quad C_2(7) = 0, \quad C_3(7) = 0, \quad C_4(7) = 0, \quad C_5(7) = 0 \end{array}$$

4279 Lagrange's interpolation implies that a polynomial of degree k , that is, that zero on $k + 1$ points
4280 has to be the zero polynomial. Since our polynomials are of degree 1 and determined on 2
4281 points, we therefore know that the only non-zero polynomials in our QAP are A_2 , A_5 , B_3 , B_4 ,
4282 C_1 and C_5 , and that we can use Lagrange's interpolation to compute them.

To compute A_2 we note that the set S in our version of Lagrange's method is given by $S = \{(x_0, y_0), (x_1, y_1)\} = \{(5, 1), (7, 0)\}$. Using this set we get:

$$\begin{aligned} A_2(x) &= y_0 \cdot l_0 + y_1 \cdot l_1 \\ &= y_0 \cdot \left(\frac{x - x_1}{x_0 - x_1} \right) + y_1 \cdot \left(\frac{x - x_0}{x_1 - x_0} \right) = 1 \cdot \left(\frac{x - 7}{5 - 7} \right) + 0 \cdot \left(\frac{x - 5}{7 - 5} \right) \\ &= \frac{x - 7}{-2} = \frac{x - 7}{11} && \# 11^{-1} = 6 \\ &= 6(x - 7) = 6x + 10 && \# -7 = 6 \text{ and } 6 \cdot 6 = 10 \end{aligned}$$

To compute A_5 , we note that the set S in our version of Lagrange's method is given by $S = \{(x_0, y_0), (x_1, y_1)\} = \{(5, 0), (7, 1)\}$. Using this set we get:

$$\begin{aligned}
 A_5(x) &= y_0 \cdot l_0 + y_1 \cdot l_1 \\
 &= y_0 \cdot \left(\frac{x - x_1}{x_0 - x_1}\right) + y_1 \cdot \left(\frac{x - x_0}{x_1 - x_0}\right) = 0 \cdot \left(\frac{x - 7}{5 - 7}\right) + 1 \cdot \left(\frac{x - 5}{7 - 5}\right) \\
 &= \frac{x - 5}{2} \quad \# 2^{-1} = 7 \\
 &= 7(x - 5) = 7x + 4 \quad \# -5 = 8 \text{ and } 7 \cdot 8 = 4
 \end{aligned}$$

Using Lagrange's interpolation, we can deduce that $A_2 = B_3 = C_5$ as well as $A_5 = B_4 = C_1$, since they are polynomials of degree 1 that evaluate to same values on 2 points. Using this, we get the following set of polynomials

$A_0(x) = 0$	$B_0(x) = 0$	$C_0(x) = 0$
$A_1(x) = 0$	$B_1(x) = 0$	$C_1(x) = 7x + 4$
$A_2(x) = 6x + 10$	$B_2(x) = 0$	$C_2(x) = 0$
$A_3(x) = 0$	$B_3(x) = 6x + 10$	$C_3(x) = 0$
$A_4(x) = 0$	$B_4(x) = 7x + 4$	$C_4(x) = 0$
$A_5(x) = 7x + 4$	$B_5(x) = 0$	$C_5(x) = 6x + 10$

We can invoke Sage to verify our computation. In sage every polynomial ring has a function `lagrange_polynomial` that takes the defining points as inputs and the associated Lagrange polynomial as output.

```

sage: F13 = GF(13)
sage: F13t.<t> = F13[]
sage: T = F13t((t-5)*(t-7))
sage: A2 = F13t.lagrange_polynomial([(5,1),(7,0)])
sage: A5 = F13t.lagrange_polynomial([(5,0),(7,1)])
sage: T == F13t(t^2 + t + 9)
True
sage: A2 == F13t(6*t + 10)
True
sage: A5 == F13t(7*t + 4)
True

```

Combining this computation with the target polynomial we derived earlier, a quadratic arithmetic program associated to the rank-1 constraint system R_{3, fac_zk} is given by

$$\begin{aligned}
 QAP(R_{3, fac_zk}) &= \{x^2 + x + 9, \\
 &\quad \{0, 0, 6x + 10, 0, 0, 7x + 4\}, \{0, 0, 0, 6x + 10, 7x + 4, 0\}, \{0, 7x + 4, 0, 0, 0, 6x + 10\}\}
 \end{aligned}$$

QAP Satisfiability One of the major points of quadratic arithmetic programs in proofing systems is that solutions of their associated rank-1 constraints systems are in 1:1 correspondence with certain polynomials P such that P is divisible by the target polynomial T of the QAP if and only if the solution is a solution. Verifying solutions to the R1CS and hence, checking proper circuit execution is then achievable by polynomial division of P by T .

To be more specific, let R be some rank-1 constraints system with associated assignment variables $(I_1, \dots, I_n; W_1, \dots, W_m)$ and let $QAP(R)$ be a quadratic arithmetic program of R . Then

clarify
language

the tuple $(I_1, \dots, I_n; W_1, \dots, W_m)$ is a solution to the R1CS if and only if the following polynomial is divisible by the target polynomial T :

$$P_{(I;W)} = (A_0 + \sum_j^n I_j \cdot A_j + \sum_j^m W_j \cdot A_{n+j}) \cdot (B_0 + \sum_j^n I_j \cdot B_j + \sum_j^m W_j \cdot B_{n+j}) - (C_0 + \sum_j^n I_j \cdot C_j + \sum_j^m W_j \cdot C_{n+j}) \quad (6.13)$$

Every tuple $(I; W)$ defines a polynomial $P_{(I;W)}$, and, since each polynomial A_j , B_j and C_j is of degree $k-1$, $P_{(I;W)}$ is of degree $(k-1) \cdot (k-1) = k^2 - 2k + 1$.

To understand how quadratic arithmetic programs define formal languages, observe that every QAP over a field \mathbb{F} defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{QAP} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{true, false\} ; (I; W) \mapsto \begin{cases} true & P_{(I;W)} \text{ is divisible by } T \\ false & \text{else} \end{cases} \quad (6.14)$$

Every QAP therefore defines a formal language, and, if the QAP is associated to an R1CS, it can be shown that the two languages are equivalent. A **statement** is a membership claim “There is a word $(I; W)$ in L_{QAP} ”. A proof to this claim is therefore a polynomial $P_{(I;W)}$, which is verified by dividing $P_{(I;W)}$ by T .

Note the structural similarity to the definition of an R1CS in XXX and the different ways of computing proofs in both systems. For circuits and their associated rank-1 constraints systems, a constructive proof consists of a valid assignment of field elements to the edges of the circuit, or the variables in the R1CS. However, in the case of QAPs, a valid proof consists of a polynomial $P_{(I;W)}$.

To compute a proof for a statement in L_{QAP} given some instance I , a proofer first needs to compute a constructive proof W , e.g. by executing the circuit. With $(I; W)$ at hand, the proofer can then compute the polynomial $P_{(I;W)}$ and publish it as proof.

Verifying a constructive proof in the case of a circuit is achieved by executing the circuit, comparing the result to the given proof, and verifying the same proof in the R1CS picture means checking if the elements of the proof satisfy all equation.

In contrast, verifying a proof in the case of a QAP is done by polynomial division of the proof P by the target polynomial T of the QAP. The proof checks out if and only if P is divisible by T .

Example 123. Consider the quadratic arithmetic program $QAP(R_{3, fac_zk})$ from example XXX and its associated R1CS from example XXX. To give an intuition of how proofs in the language $L_{QAP(R_{3, fac_zk})}$ let's consider the instance $I_1 = 11$. As we know from example XXX, $(W_1, W_2, W_3, W_5) = (2, 3, 4, 6)$ is a proper witness, since $(I_1; W_1, W_2, W_3, W_5) = (11; 2, 3, 4, 6)$ is a valid circuit assignment and hence, a solution to R_{3, fac_zk} and a constructive proof for language $L_{R_{3, fac_zk}}$.

In order to transform this constructive proof into a membership proof in language $L_{QAP(R_{3, fac_zk})}$ a proofer has to use the elements of the constructive proof, to compute the polynomial $P_{(I;W)}$.

In the case of $(I_1; W_1, W_2, W_3, W_5) = (11; 2, 3, 4, 6)$, the associated proof is computed as fol-

add reference

add reference

add reference

add reference

lows:

$$\begin{aligned}
P_{(I;W)} &= (A_0 + \sum_j^n I_j \cdot A_j + \sum_j^m W_j \cdot A_{n+j}) \cdot (B_0 + \sum_j^n I_j \cdot B_j + \sum_j^m W_j \cdot B_{n+j}) - (C_0 + \sum_j^n I_j \cdot C_j + \sum_j^m W_j \cdot C_{n+j}) \\
&= (2(6x + 10) + 6(7x + 4)) \cdot (3(6x + 10) + 4(7x + 4)) - (11(7x + 4) + 6(6x + 10)) \\
&= ((12x + 7) + (3x + 11)) \cdot ((5x + 4) + (2x + 3)) - ((12x + 5) + (10x + 8)) \\
&= (2x + 5) \cdot (7x + 7) - (9x) \\
&= (x^2 + 2 \cdot 7x + 5 \cdot 7x + 5 \cdot 7) - (9x) \\
&= (x^2 + x + 9x + 9) - (9x) \\
&= x^2 + x + 9
\end{aligned}$$

4341 Given instance $I_1 = 11$ a proofer therefore provides the polynomial $x^2 + x + 9$ as proof. To verify
 4342 this proof, any verifier can then look up the target polynomial T from the QAP and divide $P_{(I;W)}$
 4343 by T . In this particular example, $P_{(I;W)}$ is equal to the target polynomial T , and hence, it is
 4344 divisible by T with $P/T = 1$. The verification therefore checks the proof.

```

4345 sage: F13 = GF(13) 639
4346 sage: F13t.<t> = F13[] 640
4347 sage: T = F13t(t^2 + t + 9) 641
4348 sage: P = F13t((2*(6*t+10)+6*(7*t+4))*(3*(6*t+10)+4*(7*t+4)) 642
4349              -(11*(7*t+4)+6*(6*t+10)))
4350 sage: P == T 643
4351 True 644
4352 sage: P % T # remainder 645
4353 0 646

```

To give an example of a false proof, consider the tuple $(I_1; W_1, W_2, W_3, W_4) = (11, 2, 3, 4, 8)$. Executing the circuit, we can see that this is not a valid assignment and not a solution to the R1CS, and hence, not a constructive knowledge proof in $L_{3.fac_zk}$. However, a proofer might use these values to construct a false proof $P_{(I;W)}$:

$$\begin{aligned}
P'_{(I;W)} &= (A_0 + \sum_j^n I_j \cdot A_j + \sum_j^m W_j \cdot A_{n+j}) \cdot (B_0 + \sum_j^n I_j \cdot B_j + \sum_j^m W_j \cdot B_{n+j}) - (C_0 + \sum_j^n I_j \cdot C_j + \sum_j^m W_j \cdot C_{n+j}) \\
&= (2(6x + 10) + 8(7x + 4)) \cdot (3(6x + 10) + 4(7x + 4)) - (8(6x + 10) + 11(7x + 4)) \\
&= 8x^2 + 6
\end{aligned}$$

Given instance $I_1 = 11$, a proofer therefore provides the polynomial $8x^2 + 6$ as proof. To verify this proof, any verifier can look up the target polynomial T from the QAP and divide $P_{(I;W)}$ by T . However, polynomial division has a remainder

$$(8x^2 + 6)/(x^2 + x + 9) = 8 + \frac{5x + 12}{x^2 + x + 9}$$

4354 This implies that $P_{(I;W)}$ is not divisible by T , and hence, the verification does not check the
 4355 proof. Any verifier can therefore show that the proof is false.

```

4356 sage: F13 = GF(13) 647
4357 sage: F13t.<t> = F13[] 648
4358 sage: T = F13t(t^2 + t + 9) 649
4359 sage: P = F13t((2*(6*t+10)+8*(7*t+4))*(3*(6*t+10)+4*(7*t+4))-( 650
4360              8*(6*t+10)+11*(7*t+4)))

```

4361	<code>sage: P == F13t(8*t^2 + 6)</code>	651
4362	<code>True</code>	652
4363	<code>sage: P % T # remainder</code>	653
4364	<code>5*t + 12</code>	654

Chapter 7

Circuit Compilers

As we have seen in the previous chapter, statements can be formalized as membership or knowledge claims in formal language, and algebraic circuits as well as rank-1 constraint systems are two practically important ways to define those languages.

However, both algebraic circuits and rank-1 constraint systems are not ideal from a developers point of view, because they deviate substantially from common programming paradigms. Writing real-world applications as circuits and the associated verification in terms of rank-1 constraint systems is at least as troublesome as writing any other low-level language like assembler code. To allow for complex statement design, it is therefore necessary to have some kind of compiler framework, capable of transforming high-level languages into arithmetic circuits and associated rank-1 constraint systems.

As we have seen in XXX as well as XXX and XXX, both arithmetic circuits and rank-1 constraint systems have a modularity property by which it is possible to synthesize complex circuits from simple ones. A basic approach taken by many circuit/R1CS compilers is therefore to provide a library of atomic and simple circuits and then define a way to combine those basic building blocks into arbitrary complex systems.

In this chapter, we provide an introduction to basic concepts of so-called **circuit compilers** and derive a toy language which we can “compile” in a pen-and-paper approach into algebraic circuits and their associated rank-1 constraint systems.

We start with a general introduction to our language, and then introduce atomic types like booleans and unsigned integers. Then we define the fundamental control flow primitives like the if-then-else conditional and the bounded loop. We will look at basic functionality primitives like elliptic curve cryptography. Primitives like these are often called **gadgets** in the literature.

7.1 A Pen-and-Paper Language

To explain basic concepts of circuit compilers and their associated high-level languages, we derive an informal toy language and associated “brain-compiler” which we name PAPER (**Pen-And-Paper Execution Rules**). PAPER allows programmers to define statements in Rust-like pseudo-code. The language is inspired by ZOKRATES and `circom`.

7.1.1 The Grammar

In PAPER, any statement is defined as an ordered list of functions, where any function has to be declared in the list before it is called in another function of that list. The last entry in a statement has to be a special function, called `main`. Functions take a list of typed parameters as inputs

add references

add references to these languages?

and compute a tuple of typed variables as output, where types are special functions that define how to transform that type into another type, ultimately transforming any type into elements of the base field where the circuit is defined over.

Any statement is parameterized over the field that the circuit will be defined on, and has additional optional parameters of unsigned type, needed to define the size of array or the counter of bounded loops. The following definition makes the grammar of a statement precise using a command line language like description:

```
statement <Name> {F:<Field> [ , <N_1: unsigned>, ... ] } {
  [fn <Name>([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...)] {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ; ...]
  fn main([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...) {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ;
}
```

Function arguments and variables are private by default, but can be declared as public by the `pub` specifier. Declaring arguments and variables as public always overwrites any previous or conflicting private declarations. Every argument, constant or variable has a type, and every type is defined as a function that transforms that type into another type:

```
type <TYPE>( t1 : <TYPE_1>) -> TYPE_2{
  let t2: TYPE_2 <== fn(TYPE_1)
  return t2
}
```

Many real-world circuit languages are based on a similar, but of course more sophisticated approach than PAPER. The purpose of PAPER is to show basic principles of circuit compilers and their associated high-level languages.

Example 124. To get a better understanding of the grammar of PAPER, the following constitutes proper high-level code that follows the grammar of the PAPER language, assuming that all types in that code have been defined elsewhere.

```
statement MOCK_CODE {F: F_43, N_1 = 1024, N_2 = 8} {
  fn foo(in_1 : F, pub in_2 : TYPE_2) -> F {
    let const c_1 : F = 0 ;
    let const c_2 : TYPE_2 = SOME_VALUE ;
    let pub out_1 : F ;
    out_1<== c_1 ;
    return out_1 ;
  } ;

  fn bar(pub in_1 : F) -> F {
    let out_1 : F ;
    out_1<==foo(in_1);
    return out_1 ;
  } ;
}
```

```

4446     } ;
4447
4448     fn main(in_1 : TYPE_1) -> (F, TYPE_2) {
4449         let const c_1 : TYPE_1 = SOME_VALUE ;
4450         let const c_2 : F = 2 ;
4451         let const c_3 : TYPE_2 = SOME_VALUE ;
4452         let pub out_1 : F ;
4453         let out_2 : TYPE_2 ;
4454         c_1 <== in_1 ;
4455         out_1 <== foo(c_2) ;
4456         out_2 <== TYPE_2 ;
4457         return (out_1, out_2) ;
4458     } ;
4459 }

```

7.1.2 The Execution Phases

In contrast to normal executable programs, programs for circuit compilers have two modes of execution. The first mode, usually called the **setup phase**, is executed in order to generate the circuit and its associated rank-1 constraint system, the latter of which is then usually used as input to some zero-knowledge proof system.

The second mode of execution is usually called the **prover phase**. In this phase, a prover usually computes a valid assignment to the circuit. Depending on the use case, this valid assignment is then either directly used as constructive proof for proper circuit execution or is transferred as input to the proof generation algorithm of some zero-knowledge proof system, where the full-sized, non hiding constructive proof is processed into a succinct proof with various levels of zero knowledge.

Modern circuit languages and their associated compilers abstract over those two phases and provide a unified **interphase** to the developer, who then writes a single program that can be used in both phases.

To give the reader a clear, conceptual distinction between the two phases, PAPER keeps them separated. Code can be “brain-compiled” during the **setup-phase** in a pen-and-paper approach into visual circuits. Once a circuit is derived, it can be executed in a **prover phase** to generate a valid assignment. The valid assignment is then interpreted as a constructive proof for a knowledge claim in the associated language.

The Setup Phase In PAPER, the task of the setup phase is to compile code in the PAPER language into a visual representation of an algebraic circuit. Deriving the circuit from the code in a pen-and-paper style is what we call **brain compiling**.

Given some statement description that adheres to the correct grammar, we start circuit development with an empty circuit, compile the main function first and then inductively compile all other functions as they are called during the process.

For every function we compile, we draw a box-node for every argument, every variable and every constant of that function. If the node represents a variable, we label it with that variable’s name, and if it represents a constant, we label it with that constant’s value. We group arguments into a subgraph labeled “inputs” and return values into a subgraph labeled “outputs”. We then group everything into a subgraph and label that subgraph with the function’s name.

After this is done, we have to do a consistency and type check for every occurrence of the

assignment operator `<==`. We have to ensure that the expression on the right side of the operator is well defined and that the types of both side match.

Then we compile the right side of every occurrence of the assignment operator `<==`. If the right side is a constant or variable defined in this function, we draw a dotted line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator. If the right side represents an argument of that function we draw a line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator.

If the right side of the `<==` operator is a function, we look into our database, find its associated circuit and draw it. If no circuit is associated to that function yet, we repeat the compilation process for that function, drawing edges from the function's argument to its input nodes and from the functions output nodes to the nodes on the right side of `<==`.

During that process, edge labels are drawn according to the rules from XXX. If the associated variable represents a private value, we use the *W* label to indicate a witness, and if it represents a public value, we use the *I* label to indicate an instance.

add reference

Once this is done, we compile all occurring types in a function, by compiling the function of each type. We do this inductively until we reach the type of the base field. Circuits have no notion of types, only of field elements; hence, every type needs to be compiled to the field type in a sequence of compilation steps.

The compilation stops once we have inductively replaced all functions by their circuits. The result is a circuit that contains many unnecessary box nodes. In a final optimization step, all box nodes that are directly linked to each other are collapsed into a single node, and all box nodes that represent the same constants are collapsed into a single node.

Of course, PAPER's brain-compiler is not properly defined in any formal manner. Its purpose is to highlight important steps that real-world compilers undergo in their setup phases.

Example 125 (A trivial Circuit). To give an intuition of how to write and compile circuits in the PAPER language, consider the following statement description:

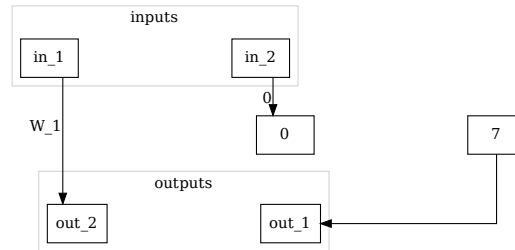
```
statement trivial_circuit {F:F_13} {
  fn main{F} (in1 : F, pub in2 : F) -> (F,F) {
    let const outc1 : F = 0 ;
    let const incl : F = 7 ;
    let out1 : F ;
    let out2 : F ;
    out1 <== incl;
    out2 <== in1;
    outc1 <== in2;
    return (out1, out2) ;
  }
}
```

To brain-compile this statement into an algebraic circuit with PAPER, we start with an empty circuit and evaluate function `main`, which is the only function in this statement.

We draw box-nodes for every argument, every constant and every variable of the function and label them with their names or values, respectively. Then we do a consistency and type check for every `<==` operator in the function. Since the circuit only wires inputs to outputs and all elements have the same type, the check is valid.

Then we evaluate the right side of the assignment operators. Since, in our case, the right side of each operator is not a function, we draw edges from the box-nodes on the right side to the associated box node on the left side. To label those edges, we use the general rules of algebraic

circuits as defined in XXX. According to those rules, every incoming edge of a sink node has a label and every outgoing edge of a source node has a label, if the node is labeled with a variable. Since nodes that represent constants are implicitly assumed to be private, and since the public specifier determines if an edge is labeled with W or I , we get the following circuit:



The Prover Phase In PAPER, a so-called **prover phase** can be executed once the setup phase has generated a circuit image from its associated high-level code. This is done by executing the circuit while assigning proper values to all input nodes of the circuit. However, in contrast to most real-world compilers, PAPER does not tell the prover how to find proper input values to a given circuit. Real-world programming languages usually provide this data by computations that are done outside of the circuit.

Example 126. Consider the circuit from example XXX. Valid assignments to this circuit are constructive proofs that the pair of inputs (S_1, S_2) is a point on the tiny-jubjub curve. However, the circuit does not provide a way to actually compute proper values for S_1 and S_2 . Any real-world system therefore needs an auxiliary computation that provides those values.

7.2 Common Programing concepts

In this section, we cover concepts that appear in almost every programming language, and see how they can be implemented in circuit compilers.

7.2.1 Primitive Types

Primitive data types like booleans, (unsigned) integers, or strings are the most basic building blocks one can expect to find in every general high-level programming language. In order to write statements as computer programs that compile into circuits, it is therefore necessary to implement primitive types as constraint systems, and define their associated operations as circuits.

In this section, we look at some common ways to achieve this. After a recapitulation of the atomic type of prime field elements, we start with an implementation of the boolean type and its associated boolean algebra as circuits. After that, we define unsigned integers based on the boolean type, and leave the implementation of signed integers as an exercise to the reader.

It should be noted, however, that while primitive data types in common programming languages (like C, Go, or Rust) have a one-to-one correspondence with objects in the computer's memory, this is not the case for most languages that compile into algebraic circuits. As we will see in the following paragraphs, common primitives like booleans or unsigned integers require many constraints and memory. Primitives different from the underlying field elements can be expensive.

The base-field type

Since both algebraic circuits and their associated rank-1 constraint systems are defined over a finite field, elements from that field are the atomic informational units in those models. In this sense, field elements $x \in \mathbb{F}$ are for algebraic circuits what bits are for computers.

In PAPER, we write F for this type and specify the actual field instance for every statement in curly brackets after the name of that statement. Two functions are associated to this type, which are induced by the **addition** and **multiplication** law in the field F . We write

$$\text{MUL} : F \times F \rightarrow F ; (x, y) \mapsto \text{MUL}(x, y) \quad (7.1)$$

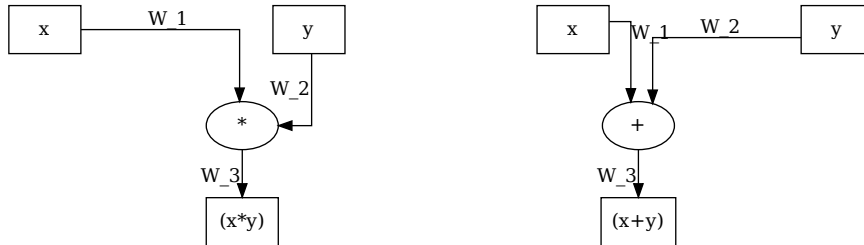
$$\text{ADD} : F \times F \rightarrow F ; (x, y) \mapsto \text{ADD}(x, y) \quad (7.2)$$

Circuit compilers have to compile these functions into algebraic gates, as explained in XXX. Every other function has to be expressed in terms of them and proper wiring.

To represent addition and multiplication in the PAPER language, we define the following two functions:

```
fn MUL(x : F, y : F) -> (MUL(x, y) : F) {}
fn ADD(x : F, y : F) -> (ADD(x, y) : F) {}
```

The compiler then compiles every occurrence of the MUL or the ADD function into the following circuits:



Example 127 (Basic gates). To give an intuition of how a real-world compiler might transform addition and multiplication in algebraic expressions into a circuit, consider the following PAPER statement:

```
statement basic_ops {F:F_13} {
  fn main(in_1 : F, pub in_2 : F) -> (out_1:F, out_2:F) {
    out_1 <== MUL(in_1, in_2) ;
    out_2 <== ADD(in_1, in_2) ;
  }
}
```

To compile this into an algebraic circuit, we start with an empty circuit and evaluate the function `main`, which is the only function in this statement.

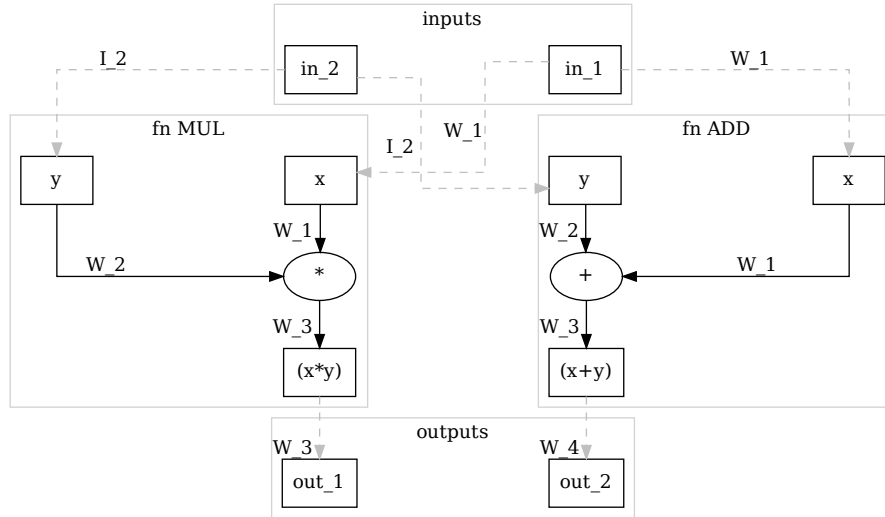
We draw an inputs subgraph containing box-nodes for every argument of the function, and an outputs subgraph containing box-nodes for every factor in the return value. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

add reference

We check the validity of every expression on the right side of every `<==` operator including a type check. In our case, every variable is of the `field` type and hence the types match the types of the `MUL` as well as the `ADD` function and the type of the left sides of `<==` operators.

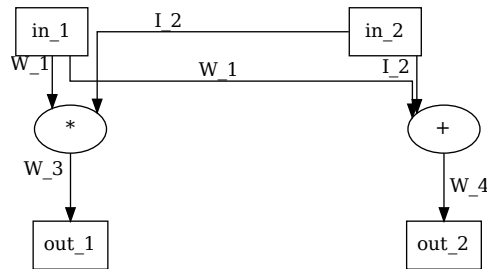
We evaluate the expressions on the right side of every `<==` operator inductively, replacing every occurrence of a function with a subgraph that represents its associated circuit.

According to `PAPER`, every occurrence of the `public` specifier overwrites the associate `private` default value. Using the appropriate edge labels we get:



Any real-world compiler might process its associated high-level language in a similar way, replacing functions, or gadgets by predefined associated circuits. This process is often followed by various optimization steps that try to reduce the number of constraints as much as possible.

In `PAPER`, we optimize this circuit by collapsing all box nodes that are directly connected to other box nodes, adhering to the rule that a variable's `public` specifier overwrites any `private` specifier. Reindexing edge labels, we get the following circuit as our pen and pencil compiler output:



Example 128 (3-factorization). Consider our 3-factorization problem from example XXX and the associated circuit $C_{3, fac_zk}(\mathbb{F}_{13})$ we provided in example XXX. To understand the process of replacing high-level functions by their associated circuits inductively, we want define a `PAPER` statement that we brain-compile into an algebraic circuit equivalent to $C_{3, fac_zk}(\mathbb{F}_{13})$. We write

add reference

add reference

```

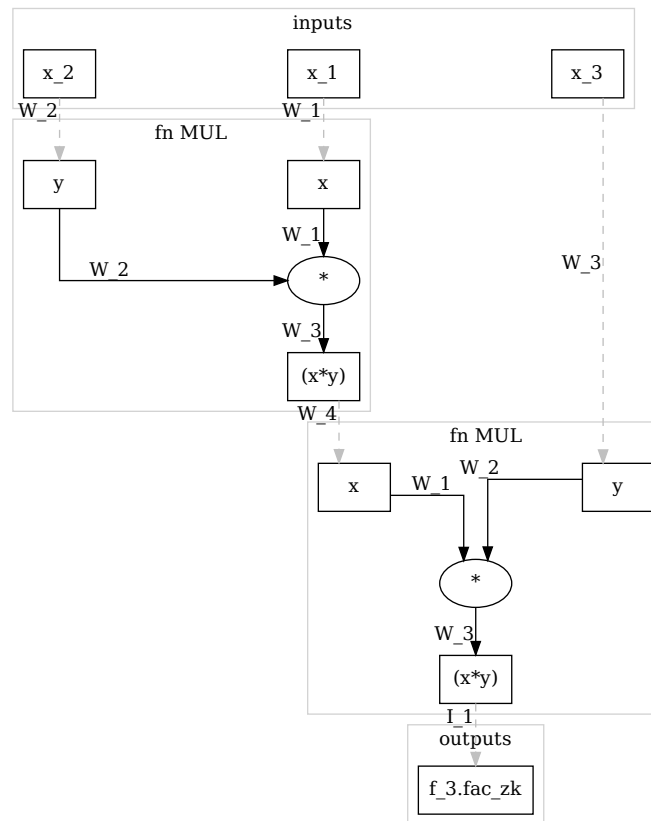
4624 statement 3_fac_zk {F:F_13} {
4625   fn main(x_1 : F, x_2 : F, x_3 : F) -> (pub 3_fac_zk : F){
4626     f_3.fac_zk <== MUL( MUL( x_1 , x_2 ) , x_3 ) ;
4627   }
4628 }

```

Using PAPER, we start with an empty circuit and then add 3 input nodes to the input subgraph as well as 1 output node to the output subgraph. All these nodes are decorated with the associated variable names. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

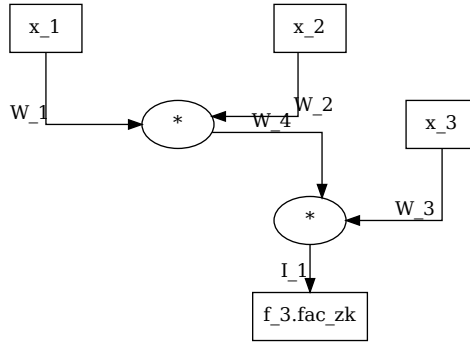
We check the validity of every expression on the right side of the single `<==` operator including a type check.

We evaluate the expressions on the right side of every `<==` operator inductively. We have two nested multiplication functions and we replace them by the associated multiplication circuits, starting with the most outer function. We get:



4638

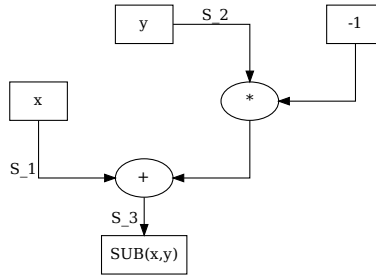
In a final optimization step, we collapse all box nodes directly connected to other box nodes, adhering to the rule that a variables `public` specifier overwrites any `private` specifier. Reindexing edge labels we get the following circuit:



4642

4643 **The Subtraction Constraint System** By definition, algebraic circuits only contain addition
 4644 and multiplication gates, and it follows that there is no single gate for field subtraction, despite
 4645 the fact that subtraction is a native operation in every field.

4646 High-level languages and their associated circuit compilers, therefore, need another way to
 4647 deal with subtraction. To see how this can be achieved, recall that subtraction is defined by addi-
 4648 tion with the additive inverse, and that the inverse can be computed efficiently by multiplication
 4649 with -1 . A circuit for field subtraction is therefore given by



4650

4651 Using the general method from XXX, the circuits associated rank-1 constraint system is given
 4652 by:

$$(S_1 + (-1) \cdot S_2) \cdot 1 = S_3 \quad (7.3)$$

4653 Any valid assignment $\{S_1, S_2, S_3\}$ to this circuit therefore enforces the value S_3 to be the differ-
 4654 ence $S_1 - S_2$.

4655 Real-world compilers usually provide a gadget or a function to abstract over this circuit
 4656 such that programers can use subtraction as if it were native to circuits. In PAPER, we define
 4657 the following subtraction function that compiles to the previous circuit:

```
4658 fn SUB(x : F, y : F) -> (SUB(x,y) : F) {
4659   constant c : F = -1 ;
4660   SUB <== ADD(x , MUL( y , c ) );
4661 }
```

4662 In the setup phase of a statement, we compile every occurrence of the SUB function into an
 4663 instance of its associated subtraction circuit, and edge labels are generated according to the
 4664 rules from XXX.

add refer-
enceadd refer-
ence

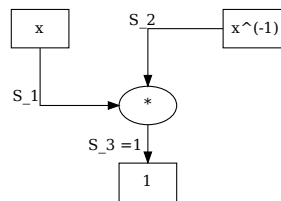
The Inversion Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field inversion, despite the fact that inversion is a native operation in every field.

If the underlying field is a prime field, one approach would be to use Fermat’s little theorem [XXX](#) to compute the multiplicative inverse inside the circuit. To see how this works, let \mathbb{F}_p be the prime field. The multiplicative inverse x^{-1} of a field element $x \in \mathbb{F}$ with $x \neq 0$ is then given by $x^{-1} = x^{p-2}$, and computing x^{p-2} in the circuit therefore computes the multiplicative inverse.

Unfortunately, real-world primes p are large and computing x^{p-2} by repeated multiplication of x with itself is infeasible. A “double and multiply” approach (as described in [XXX](#)) is faster, as it computes the power in roughly $\log_2(p)$ steps, but still adds a lot of constraints to the circuit.

Computing inverses in the circuit makes no use of the fact that inversion is a native operation in any field. A more constraints friendly approach is therefore to compute the multiplicative inverse outside of the circuit and then only enforce correctness of the computation in the circuit.

To understand how this can be achieved, observe that a field element $y \in \mathbb{F}$ is the multiplicative inverse of a field element $x \in \mathbb{F}$ if and only if $x \cdot y = 1$ in \mathbb{F} . We can use this, and define a circuit that has two inputs, x and y , and enforces $x \cdot y = 1$. It is then guaranteed that y is the multiplicative inverse of x . The price we pay is that we can not compute y by circuit execution, but auxiliary data is needed to tell any prover which value of y is needed for a valid circuit assignment. The following circuit defines the constraint



Using the general method from [XXX](#), the circuit is transformed into the following rank-1 constraint system:

$$S_1 \cdot S_2 = 1 \quad (7.4)$$

Any valid assignment $\{S_1, S_2\}$ to this circuit enforces that S_2 is the multiplicative inverse of S_1 , and, since there is no field element S_2 such that $0 \cdot S_2 = 1$, it also handles the fact that the multiplicative inverse of 0 is not defined in any field.

Real-world compilers usually provide a gadget or a function to abstract over this circuit, and those functions compute the inverse x^{-1} as part of their witness generation process. Programmers then don’t have to care about providing the inverse as auxiliary data to the circuit. In [PAPER](#), we define the following inversion function that compiles to the previous circuit:

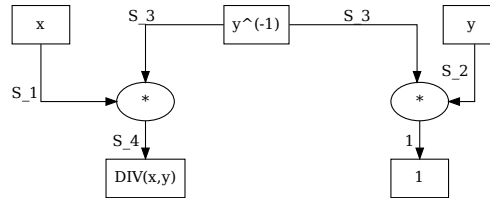
```
fn INV(x : F, y : F) -> (x_inv : F) {
  constant c : F = 1 ;
  c <== MUL( x , y ) ;
  x_inv <== y ;
}
```

As we see, this functions takes two inputs, the field value and its inverse. It therefore does not handle the computation of the inverse by itself. This is to keep [PAPER](#) as simple as possible.

In the setup phase, we compile every occurrence of the INV function into an instance of the inversion circuit [XXX](#), and edge labels are generated according to the rules from [XXX](#).

The Division Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field division, despite the fact that division is a native operation in every field.

Implementing division as a circuit, we use the fact that division is multiplication with the multiplicative inverse. We therefore define division as a circuit using the inversion circuit and constraint system from the previous paragraph. Expensive inversion is computed outside of the circuit and then provided as circuit input. We get



Using the method from XXX, we transform this circuit into the following rank-1 constraint system:

$$\begin{aligned} S_2 \cdot S_3 &= 1 \\ S_1 \cdot S_3 &= S_4 \end{aligned}$$

Any valid assignment $\{S_1, S_2, S_3, S_4\}$ to this circuit enforces S_4 to be the field division of S_1 by S_2 . It handles the fact that division by 0 is not defined, since there is no valid assignment in case $S_2 = 0$.

In PAPER, we define the following division function that compiles to the previous circuit:

```
fn DIV(x : F, y : F, y_inv : F) -> (DIV : F) {
  DIV <== MUL( x , INV( y, y_inv ) ) ;
}
```

In the setup phase, we compile every occurrence of the binary INV operator into an instance of the inversion circuit.

Exercise 44. Let F be the field \mathbb{F}_5 of modular 5 arithmetics from example XXX. Brain compile the following PAPER statement into an algebraic circuit:

```
statement STUPID_CIRC {F: F_5} {
  fn foo(in_1 : F, in_2 : F)->(out_1 : F, out_2 : F){
    constant c_1 : F = 3 ;
    out_1<== ADD( MUL( c_1 , in_1 ) , in_1 ) ;
    out_2<== INV( c_1 , in_2 ) ;
  } ;

  fn main(in_1 : F, in_2 : F)->(out_1 : F, out_2 : TYPE_2){
    constant (c_1,c_2) : (F,F) = (3,2) ;
    (out_1,out_2) <== foo(in_1, in_2) ;
  } ;
}
```

Exercise 45. Consider the tiny-jubjub curve from example XXX and its associated circuit XXX. Write a statement in PAPER that brain-compiles the statement into a circuit equivalent to the one derived in XXX, assuming that curve points are instances and every other assignment is a witness.

4738 *Exercise 46.* Let $F = \mathbb{F}_{13}$ be the modular 13 prime field and $x \in F$ some field element. Define a
 4739 statement in PAPER such that given instance x a field element $y \in F$ is a witness for the statement
 4740 if and only if y is the square root of x .

4741 Brain compile the statement into a circuit and derive its associated rank-1 constraint system.
 4742 Consider the instance $x = 9$ and compute a constructive proof for the statement.

4743 The boolean Type

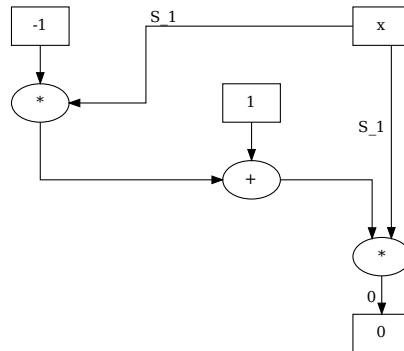
4744 booleans are a classical primitive type, implemented by virtually every higher programing lan-
 4745 guage. It is therefore important to implement booleans in circuits. One of the most common
 4746 ways to do this is by interpreting the additive and multiplicative neutral element $\{0, 1\} \subset \mathbb{F}$ as
 4747 the two boolean values such that 0 represents *false* and 1 represents *true*. boolean operators
 4748 like *and*, *or*, or *xor* are then expressible as algebraic computations inside \mathbb{F} .

4749 Representing booleans this way is convenient, because the elements 0 and 1 are defined in
 4750 any field. The representation is therefore independent of the actual field in consideration.

4751 To fix boolean algebra notation, we write 0 to represent *false* and 1 to represent *true*, and
 4752 we write \wedge to represent the boolean AND as well as \vee to represent the boolean OR operator.
 4753 The boolean NOT operator is written as \neg .

4754 **The boolean Constraint System** To represent booleans by the additive and multiplicative
 4755 neutral elements of a field, a constraint is required to actually enforce variables of boolean type
 4756 to be either 1 or 0. In fact, many of the following circuits that represent boolean functions are
 4757 only correct under the assumption that their input variables are constrained to be either 0 or 1.
 4758 Not constraining boolean variables is a common problem in circuit design.

4759 In order to constrain an arbitrary field element $x \in \mathbb{F}$ to be 1 or 0, the key observation is
 4760 that the equation $x \cdot (1 - x) = 0$ has only two solutions 0 and 1 in any field. Implementing this
 4761 equation as a circuit therefore generates the correct constraint:



4762

Using the method from XXX, we transform this circuit into the following rank-1 constraint system:

$$S_1 \cdot (1 - S_1) = 0$$

add refer-
ence

4763 Any valid assignment $\{S_1\}$ to this circuit enforces S_1 to be either 0 or 1.

4764 Some real-world circuit compilers like ZOKRATES or BELLMAN are typed, while others like
 4765 circom are not. However, all of them have their way of dealing with the binary constraint. In
 4766 PAPER, we define the following boolean type that compiles to the previous circuit:

```

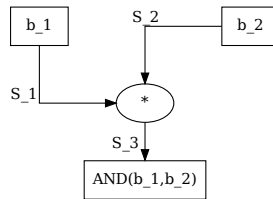
4767 type BOOL(b : BOOL) -> (x : F) {
4768     constant c1 : F = 0 ;
4769     constant c2 : F = 1 ;
4770     constant c3 : F = -1 ;
4771     c1 <== MUL( x , ADD( c2 , MUL( x , c3 ) ) ) ;
4772     x <== b ;
4773 }

```

4774 In the setup phase of a statement, we compile every occurrence of a variable of boolean type
 4775 into an instance of its associated boolean circuit.

4776 **The AND operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 4777 constrained to represent boolean variables, we want to find a circuit that computes the logical
 4778 **and** operator $AND(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, AND(b_1, b_2)$ to
 4779 satisfy the constraint system if and only if $b_1 \wedge b_2 = AND(b_1, b_2)$ holds true.

4780 The key insight here is that given three boolean constraint variables b_1, b_2 and b_3 , the equa-
 4781 tion $b_1 \cdot b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \wedge b_2 = b_3$ is satisfied in boolean
 4782 algebra. The logical operator \wedge is therefore implementable in \mathbb{F} by field multiplication of its
 4783 arguments and the following circuit computes the \wedge operator in \mathbb{F} , assuming all inputs are re-
 4784 stricted to be 0 or 1:



4785

4786 The associated rank-1 constraint system can be deduced from the general process XXX and
 4787 consists of the following constraint

$$S_1 \cdot S_2 = S_3 \quad (7.5)$$

add refer-
ence

4788 Common circuit languages typically provide a gadget or a function to abstract over this circuit
 4789 such that programers can use the \wedge operator without caring about the associated circuit. In
 4790 PAPER, we define the following function that compiles to the \wedge -operator's circuit:

```

4791 fn AND(b_1 : BOOL, b_2 : BOOL) -> AND(b_1, b_2) : BOOL{
4792     AND(b_1, b_2) <== MUL( b_1 , b_2 ) ;
4793 }

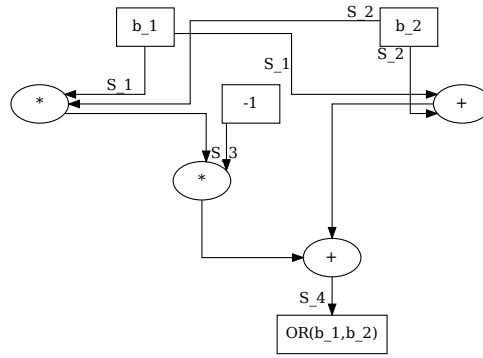
```

4794 In the setup phase of a statement, we compile every occurrence of the AND function into an
 4795 instance of its associated \wedge -operator's circuit.

4796 **The OR operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 4797 constrained to represent boolean variables, we want to find a circuit that computes the logical
 4798 **or** operator $OR(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, OR(b_1, b_2)$ to satisfy
 4799 the constraint system if and only if $b_1 \vee b_2 = OR(b_1, b_2)$ holds true.

4800 Assuming that three variables b_1, b_2 and b_3 are boolean constraint, the equation $b_1 + b_2 - b_1 \cdot$
 4801 $b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \vee b_2 = b_3$ is satisfied in boolean algebra.
 4802 The logical operator \vee is therefore implementable in \mathbb{F} by the following circuit, assuming all
 4803 inputs are restricted to be 0 or 1:

"constraints"
or "con-
strained"?



4804

The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$S_1 \cdot S_2 = S_3$$

$$(S_1 + S_2 - S_3) \cdot 1 = S_4$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the \vee operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the \vee -operator's circuit:

```
4808 fn OR(b_1 : BOOL, b_2 : BOOL) -> OR(b_1,b_2) : BOOL{
4809     constant c1 : F = -1 ;
4810     OR(b_1,b_2) <== ADD(ADD(b_1,b_2),MUL(c1,MUL(b_1,b_2))) ;
4811 }
```

In the setup phase of a statement, we compile every occurrence of the OR function into an instance of its associated \vee -operator's circuit.

Exercise 47. Let \mathbb{F} be a finite field and let b_1 as well as b_2 two boolean constraint variables from \mathbb{F} . Show that the equation $OR(b_1, b_2) = 1 - (1 - b_1) \cdot (1 - b_2)$ holds true.

Use this equation to derive an algebraic circuit with ingoing variables b_1 and b_2 and outgoing variable $OR(b_1, b_2)$ such that b_1 and b_2 are boolean **constraint** and the circuit has a valid assignment, if and only if $OR(b_1, b_2) = b_1 \vee b_2$.

"constraints" or "constrained"?

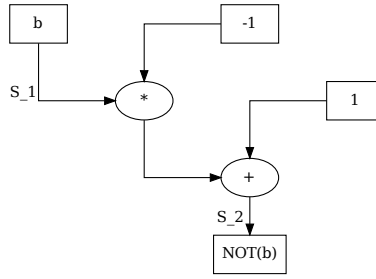
Use the technique from XXX to transform this circuit into a rank-1 constraint system and find its full solution set. Define a PAPER function that brain-compiles into the circuit.

add reference

The NOT operator constraint system Given a field element b from \mathbb{F} that is constrained to represent a boolean variable, we want to find a circuit that computes the logical **NOT** operator $NOT(b)$ as well as its associated RICS that enforces $b, NOT(b)$ to satisfy the constraint system if and only if $\neg b = NOT(b)$ holds true.

Assuming that two variables b_1 and b_2 are boolean **constraint**, the equation $(1 - b_1) = b_2$ is satisfied in \mathbb{F} if and only if the equation $\neg b_1 = b_2$ is satisfied in boolean algebra. The logical operator \neg is therefore implementable in \mathbb{F} by the following circuit, assuming all inputs are restricted to be 0 or 1:

"constraints" or "constrained"?



4829

The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$(1 - S_1) \cdot 1 = S_2$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the \neg operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the \neg -operator's circuit:

```

4833 fn NOT(b : BOOL -> NOT(b) : BOOL{
4834   constant c1 = 1 ;
4835   constant c2 = -1 ;
4836   NOT(b_1) <== ADD( c1 , MUL( c2 , b ) ) ;
4837 }

```

In the setup phase of a statement, we compile every occurrence of the NOT function into an instance of its associated \neg -operator's circuit.

Exercise 48. Let \mathbb{F} be a finite field. Derive the algebraic circuit and associated rank-1 constraint system for the following operators: NOR, XOR, NAND, EQU.

Modularity As we have seen in XXX and XXX, both algebraic circuits and R1CS have a modularity property, and as we have seen in this section, all basic boolean functions are expressible in circuits. Combining those two properties, show that it is possible to express arbitrary boolean functions as algebraic circuits.

add reference

This shows that the expressiveness of algebraic circuits and therefore rank-1 constraint systems is as general as the expressiveness of boolean circuits. An important implication is that the languages $L_{R1CS-SAT}$ and $L_{Circuit-SAT}$ as defined in XXX, are as general as the famous language L_{3-SAT} , which is known to be \mathcal{NP} -complete.

add reference

Example 129. To give an example of how a compiler might construct complex boolean expressions in algebraic circuits from simple ones and how we derive their associated rank-1 constraint systems, let's look at the following PAPER statement:

```

4853 statement BOOLEAN_STAT {F: F_p} {
4854   fn main(b_1:BOOL,b_2:BOOL,b_3:BOOL,b_4:BOOL )-> pub b_5:BOOL {
4855     b_5 <== AND( OR( b_1 , b_2 ) , AND( b_3 , NOT( b_4 ) ) ) ;
4856   } ;
4857 }

```

add reference

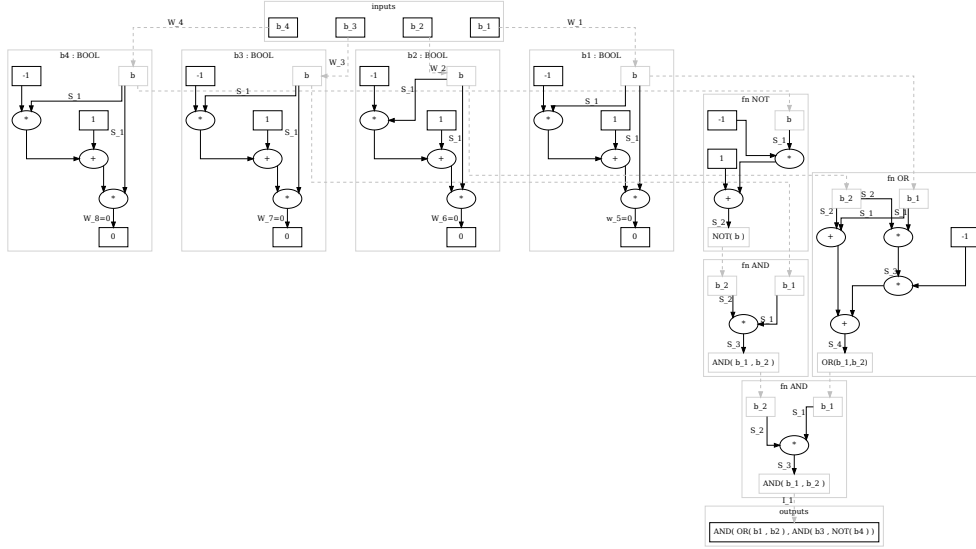
The code describes a circuit that takes four private inputs b_1, b_2, b_3 and b_4 of boolean type and computes a public output b_5 such that the following boolean expression holds true:

$$(b_1 \vee b_2) \wedge (b_3 \wedge \neg b_4) = b_5$$

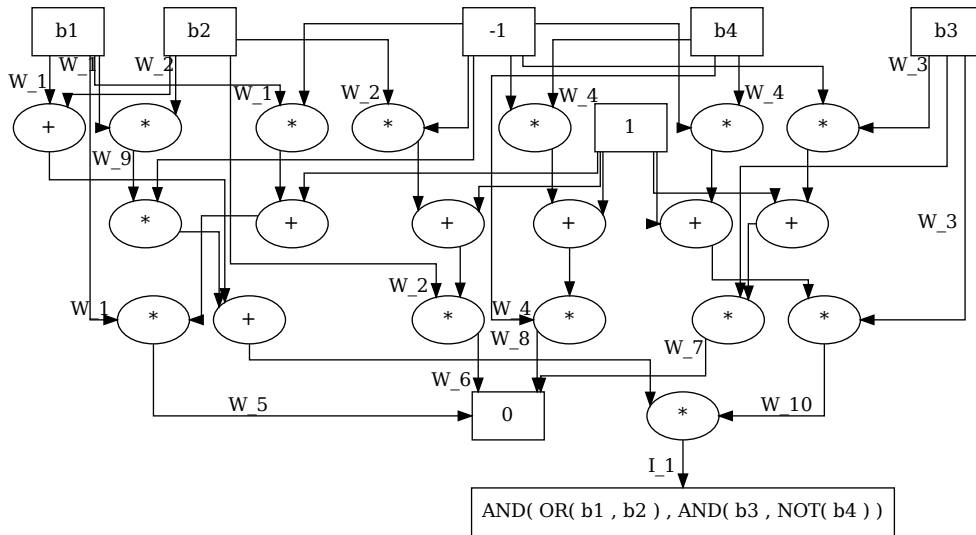
During a setup-phase, a circuit compiler transforms this high-level language statement into a circuit and associated rank-1 constraint systems and hence defines a language $L_{BOOLEAN_STAT}$.

To see how this might be achieved, we use PAPER as an example to execute the setup-phase and compile `BOOLEAN_STAT` into a circuit. Taking the definition of the boolean constraint `XXX` as well as the definitions of the appropriate boolean operators into account, we get the following circuit:

add reference



Simple optimization then collapses all box-nodes that are directly linked and all box nodes that represent the same constants. After relabeling the edges, the following circuit represents the circuit associated to the `BOOLEAN_STAT` statement:



Given some public input I_1 from \mathbb{F}_{13} , a valid assignment to this circuit consists of private inputs W_1, W_2, W_3, W_4 from \mathbb{F}_{13} such that the equation $I_1 = (W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4)$ holds true. In addition, a valid assignment also has to contain private inputs W_5, W_6, W_7, W_8, W_9 and W_{10} , which can be derived from circuit execution. The inputs W_5, \dots, W_8 ensure that the first four

4873 private inputs are either 0 or 1 but not any other field element, and the others enforce the boolean
4874 operations in the expression.

To compute the associated RICS, we can use the general method from XXX and look at every labeled outgoing edge not coming from a source node. Declaring the edges coming from input nodes as well as the edge going to the single output node as public, and every other edge as private input. In this case we get:

add refer-
ence

$$\begin{aligned}
 W_5 : W_1 \cdot (1 - W_1) &= 0 && \text{boolean constraints} \\
 W_6 : W_2 \cdot (1 - W_2) &= 0 \\
 W_7 : W_3 \cdot (1 - W_3) &= 0 \\
 W_8 : W_4 \cdot (1 - W_4) &= 0 \\
 W_9 : W_1 \cdot W_2 &= W_9 && \text{first OR-operator constraint} \\
 W_{10} : W_3 \cdot (1 - W_4) &= W_{10} && \text{AND(.,NOT(.))-operator constraints} \\
 I_1 : (W_1 + W_2 - W_9) \cdot W_{10} &= I_1 && \text{AND-operator constraints}
 \end{aligned}$$

4875 The reason why this RICS only contains a single constraint for the multiplication gate in the
4876 OR-circuit, while the general definition XXX requires two constraints, is that the second con-
4877 straint in XXX only appears because the final addition gate is connected to an output node. In
4878 this case, however, the final addition gate from the OR-circuit is enforced in the left factor of
4879 the I_1 constraint. Something similar holds true for the negation circuit.

add refer-
ence

add refer-
ence

During a prover-phase, some public instance I_5 must be given. To compute a constructive proof for the statement of the associated languages with respect to instance I_5 , a prover has to find four boolean values W_1, W_2, W_3 and W_4 such that

$$(W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4) = I_5$$

4880 holds true. In our case neither the circuit nor the PAPER statement specifies how to find those
4881 values, and it is a problem that any prover has to solve outside of the circuit. This might or
4882 might not be true for other problems, too. In any case, once the prover found those values, they
4883 can execute the circuit to find a valid assignment.

To give a concrete example, let $I_1 = 1$ and assume $W_1 = 1, W_2 = 0, W_3 = 1$ and $W_4 = 0$. Since $(1 \vee 0) \wedge (1 \wedge \neg 0) = 1$, those values satisfy the problem and we can use them to execute the circuit. We get

$$\begin{aligned}
 W_5 &= W_1 \cdot (1 - W_1) = 0 \\
 W_6 &= W_2 \cdot (1 - W_2) = 0 \\
 W_7 &= W_3 \cdot (1 - W_3) = 0 \\
 W_8 &= W_4 \cdot (1 - W_4) = 0 \\
 W_9 &= W_1 \cdot W_2 = 0 \\
 W_{10} &= W_3 \cdot (1 - W_4) = 1 \\
 I_1 &= (W_1 + W_2 - W_9) \cdot W_{10} = 1
 \end{aligned}$$

4884 A constructive proof of knowledge of a witness, for instance, $I_1 = 1$, is therefore given by the
4885 tuple $P = (W_5, W_6, W_7, W_8, W_9, W_{10}) = (0, 0, 0, 0, 0, 1)$.

Arrays

The `array` type represents a fixed-size collection of elements of equal type, each selectable by one or more indices that can be computed at run time during program execution.

Arrays are a classical type, implemented by many higher programming languages that compile to circuits or rank-1 constraint systems. However, most high-level circuit languages support **static** arrays, i.e., arrays whose length is known at compile time only.

The most common way to compile arrays to circuits is to transform any array of a given type τ and size N into N circuit variables of type τ . Arrays are therefore **syntactic sugar**, that is, parts of the formal language that makes the code easier for humans to read, which the compiler transforms into input nodes, much like any other variable. In PAPER, we define the following array type:

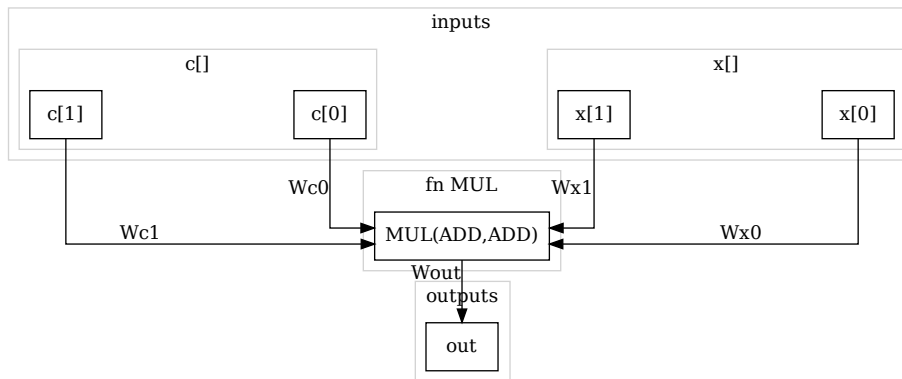
```
type <Name>: <Type>[N : unsigned] -> (Type, ...) {
  return (<Name>[0], ...)
}
```

In the setup phase of a statement, we compile every occurrence of an array of size N that contains elements of type `Type` into N variables of type `Type`.

Example 130. To give an intuition of how a real-world compiler might transform arrays into circuit variables, consider the following PAPER statement:

```
statement ARRAY_TYPE {F: F_5} {
  fn main(x: F[2]) -> F {
    let constant c: F[2] = [2, 4] ;
    let out: F <== MUL(ADD(x[1], c[0]), ADD(x[0], c[1])) ;
    return out ;
  } ;
}
```

During a setup phase, a circuit compiler might then replace any occurrence of the array type by a tuple of variables of the underlying type, and then use those variables in the circuit synthesis process instead. To see how this can be achieved, we use PAPER as an example. Abstracting over the sub-circuit of the computation, we get the following circuit:



The Unsigned Integer Type

Unsigned integers of size N , where N is usually a power of two, represent non-negative integers in the range $0 \dots 2^N - 1$. They have a notion of addition, subtraction and multiplication, defined

by modular 2^N arithmetics. If some N is given, we write uN for the associated type.

The uN Constraint System Many high-level circuit languages define the the various uN types as arrays of size N , where each element is of boolean type. This is similar to their representation on common computer hardware and allows for efficient and straightforward definition of common operators, like the various **shift**, or logical operators.

shift

If some unsigned integer N is known at compile time in PAPER, we define the following uN type:

```

type uN -> BOOL[N] {
  let base2 : BOOL[N] <== BASE_2 (uN) ;
  return base2 ;
}

```

To enforce an N -tuple of field elements (b_0, \dots, b_{N-1}) to represent an element of type uN we therefore need N boolean constraints

$$\begin{aligned}
 S_0 \cdot (1 - S_0) &= 0 \\
 S_1 \cdot (1 - S_1) &= 0 \\
 &\dots \\
 S_{N-1} \cdot (1 - S_{N-1}) &= 0
 \end{aligned}$$

In the setup phase of a statement, we compile every occurrence of the uN type by a size N array of boolean type. During a=the prover phase, actual elements of the uN type are first transformed into binary representation and then this binary representation is assigned to the boolean array that represents the uN type.

Remark 4. Representing the uN type as boolean arrays is conceptually clean and works over generic base fields. However, representing unsigned integers in this way requires a lot of space as every bit is represented as a field element and if the base field is large, those field elements require considerable space in hardware.

It should be noted that, in some cases, there is another, more space- and constraint-efficient approach for representing unsigned integers that can be used whenever the underlying base field is sufficiently large. To understand this, recall that addition and multiplication in a prime field \mathbb{F}_p is equal to addition and multiplication of integers, as long as the sum or the product does not exceed the modulus p . It is therefore possible to represent the uN type inside the base-field type whenever N is small enough. In this case, however, care has to be taken to never overflow the modulus. It is also important to make sure that, in the case of subtraction, the subtrahend is never larger than the minuend.

Example 131. To give an intuition of how a real-world compiler might transform unsigned integers into circuit variables, consider the following PAPER statement:

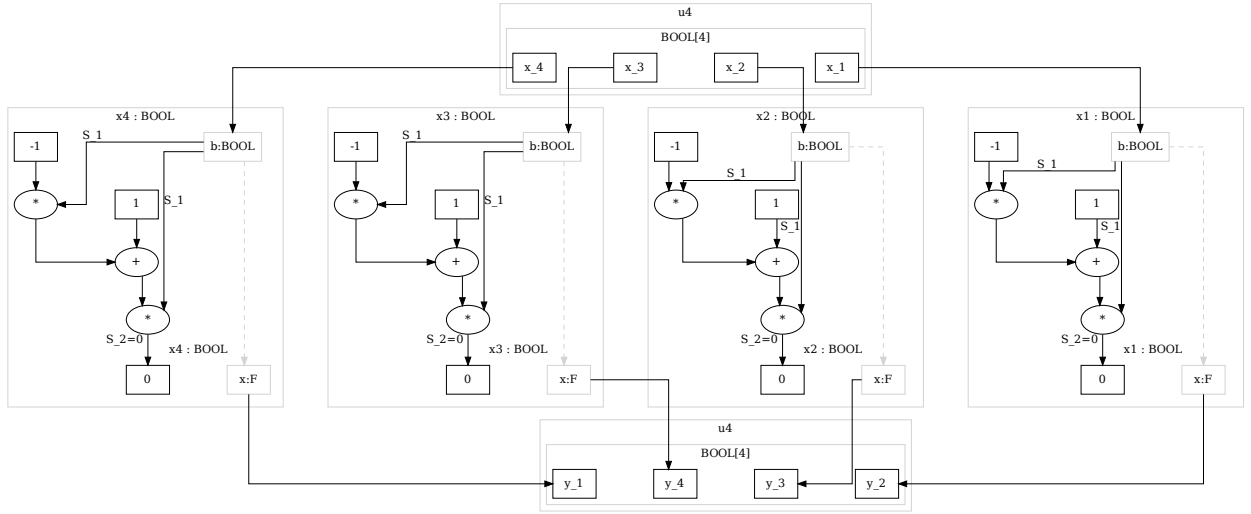
```

statement RING_SHIFT{F: F_p, N=4} {
  fn main(x: uN) -> uN {
    let y:uN <== [x[1], x[2], x[3], x[0]] ;
    return y ;
  } ;
}

```

During the setup-phase, a circuit compiler might then replace any occurrence of the uN type by N variables of `boolean` type. Using the definition of booleans, each of these variables is

then transformed into the `field` type and a boolean constraint system. To see how this can be achieved, we use PAPER as an example and get the following circuit:



During the prover phase, the function `main` is called with an actual input of `u4` type, say $x=14$. The high-level language then has to transform the decimal value 14 into its 4-bit binary representation $14_2 = (0, 1, 1, 1)$ outside of the circuit. Then the array of field values $x[4] = [0, 1, 1, 1]$ is used as an input to the circuit. Since all 4 field elements are either 0 or 1, the four boolean constraints are satisfiable and the output is an array of the four field elements $[1, 1, 1, 0]$, which represents the `u4` element 7.

The Unigned Integer Operators Since elements of `uN` type are represented as boolean arrays, shift operators are implemented in circuits simply by rewiring the boolean input variables to the output variables accordingly.

Logical operators, like AND, OR, or NOT are defined on the `uN` type by invoking the appropriate boolean operators bitwise to every bit in the boolean array that represents the `uN` element.

Addition and multiplication can be represented similarly to how machines represent those operations. Addition can be implemented by first defining the **full adder** circuit and then combining N of these circuits into a circuit that adds two elements from the `uN` type.

Exercise 49. Let $F = \mathbb{F}_{13}$ and $N=4$ be fixed. Define circuits and associated R1CS for the left and right **bishift** operators $x \ll 2$ as well as $x \gg 2$ that operate on the `uN` type. Execute the associated circuit for $x : u4 = 11$.

Exercise 50. Let $F = \mathbb{F}_{13}$ and $N=2$ be fixed. Define a circuit and associated R1CS for the addition operator $\text{ADD} : F \times F \rightarrow F$. Execute the associated circuit to compute $\text{ADD}(2, 7)$.

Exercise 51. Brain compile the following PAPER code into a circuit and derive the associated R1CS.

```
statement MASK_MERGE {F:F_5, N=4} {
  fn main(pub a : uN, pub b : uN) -> F {
    let constant mask : uN = 10 ;
    let r : uN <== XOR(a, AND(XOR(a,b), mask)) ;
    return r ;
  }
}
```

```

4987     }
4988 }

```

4989 Let L_{mask_merge} be the language defined by the circuit. Provide a constructive knowledge proof
 4990 in L_{mask_merge} for the instance $I = (I_a, I_b) = (14, 7)$.

4991 7.2.2 Control Flow

4992 Most programing languages of the imperative or functional style have some notion of basic
 4993 control structures to direct the order in which instructions are evaluated. Contemporary circuit
 4994 compilers usually provide a single thread of execution and provide basic flow constructs that
 4995 implement control flow in circuits.

4996 The Conditional Assignment

4997 Writing high-level code that compiles to circuits, it is often necessary to have a way for condi-
 4998 tional assignment of values or computational output to variables.

4999 One way to realize this in many programming languages is in terms of the conditional
 5000 ternary assignment operator $?:$ that branches the control flow of a program according to some
 5001 condition and then assigns the output of the computed branch to some variable.

```

5002 variable = condition ? value_if_true : value_if_false

```

5003 In this description, `condition` is a boolean expression and `value_if_true` as well as
 5004 `value_if_false` are expressions that evaluate to the same type as `variable`.

5005 In programming languages like Rust, another way to write the conditional assignment oper-
 5006 ator that is more familiar to many programmers is given by

```

5007 variable = if condition then {
5008     value_if_true
5009 } else {
5010     value_if_false
5011 }

```

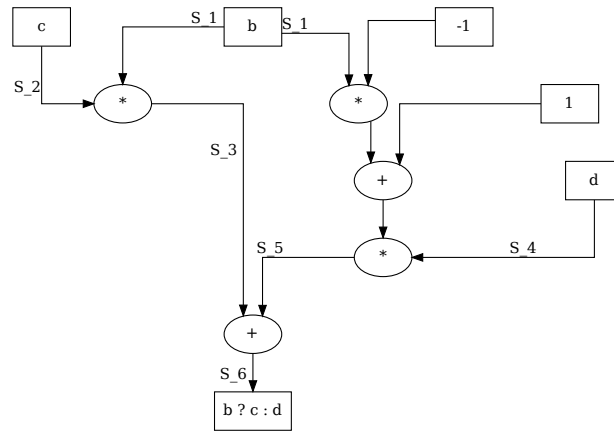
5012 In most programing languages, it is a key property of the ternary assignment operator that
 5013 the expression `value_if_true` is only evaluated if `condition` evaluates to true and the
 5014 expression `value_if_false` is only evaluated if `condition` evaluates to false. In fact,
 5015 computer programs would turn out to be very inefficient if the ternary operator would evaluate
 5016 both expressions regardless of the value of `condition`.

5017 A simple way to implement conditional assignment operator as a circuit can be achieved
 5018 if the requirement that only one branch of the conditional operator is executed is dropped. To
 5019 see that, let b , c and d be field elements such that b is a boolean constraint. In this case, the
 5020 following equation enforces a field element x to be the result of the conditional assignment
 5021 operator:

$$x = b \cdot c + (1 - b) \cdot d \quad (7.6)$$

5022 Expressing this equation in terms of the addition and multiplication operators from XXX, we
 5023 can flatten it into the following algebraic circuit:

add refer-
ence



5024

5025 Note that, in order to compute a valid assignment to this circuit, both S_2 as well as S_4 are
 5026 necessary. If the inputs to the nodes c and d are circuits themselves, both circuits need valid
 5027 assignments and therefore have to be executed. As a consequence, this implementation of
 5028 the conditional assignment operator has to execute all branches of all circuits, which is very
 5029 different from the execution of common computer programs and contributes to the increased
 5030 computational effort any prover has to invest, in contrast to the execution in other programming
 5031 models.

We can use the general technique from XXX to derive the associated rank-1 constraint system of the conditional assignment operator. We get

add reference

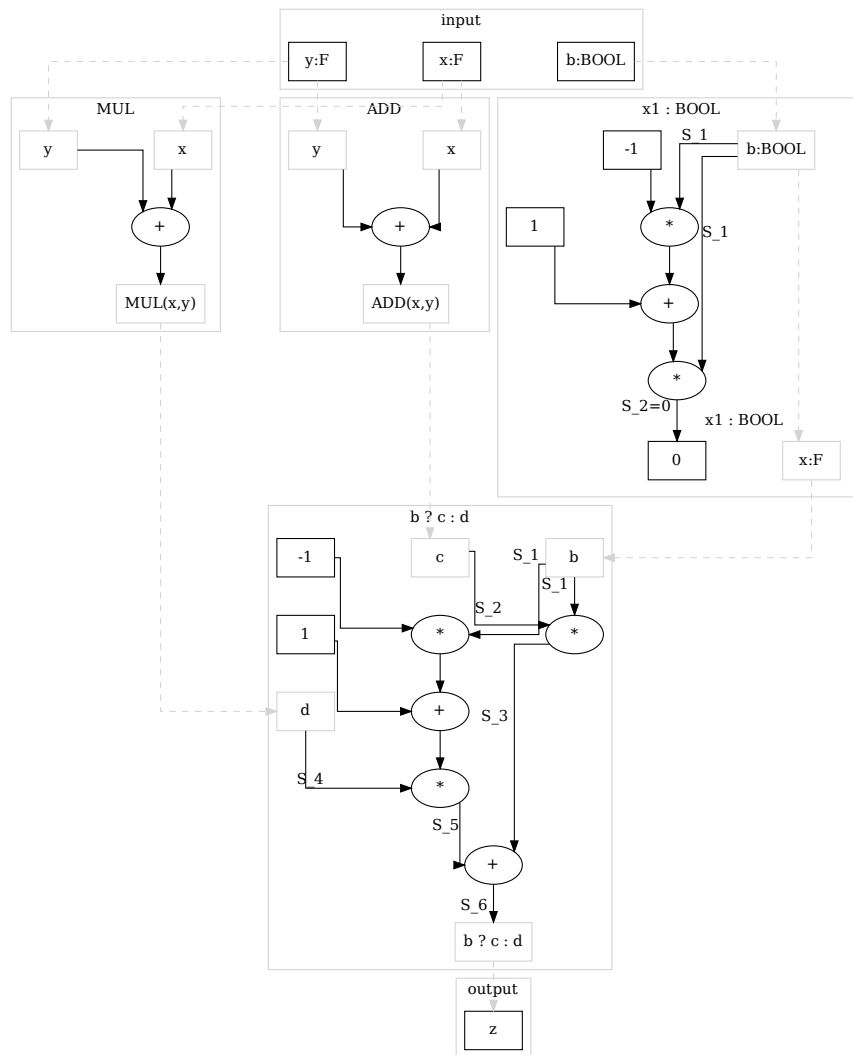
$$\begin{aligned} S_1 \cdot S_2 &= S_3 \\ (1 - S_1) \cdot S_4 &= S_5 \\ (S_3 + S_5) \cdot 1 &= S_6 \end{aligned}$$

5032 *Example 132.* To give an intuition of how a real-world circuit compiler might transform any
 5033 high-level description of the conditional assignment operator into a circuit, consider the follow-
 5034 ing PAPER code:

```

5035 statement CONDITIONAL_OP {F:F_p} {
5036   fn main(x : F, y : F, b : BOOL) -> F {
5037     let z : F <== if b then {
5038       ADD(x,y)
5039     } else {
5040       MUL(x,y)
5041     } ;
5042     return z ;
5043   }
5044 }
```

5045 Brain compiling this code into a circuit, we first draw box nodes for all input and output vari-
 5046 ables, and then transform the boolean type into the field type together with its associated con-
 5047 straint. Then we evaluate the assignments to the output variables. Since the conditional assign-
 5048 ment operator is the top level function, we draw its circuit and then draw the circuits for both
 5049 conditional expressions. We get:



5050

5051 **Loops**

5052 In many programming languages, various loop control structures are defined that allow devel-
 5053 opers to execute expressions with a specified number of repetitions or arguments. In particular,
 5054 it is often possible to implement unbounded loops like the

5055 **while** true do { }
 5056

5057 structure, or loop structure, where the number of executions depends on execution inputs
 5058 and is therefore unknown at compile time.

5059 In contrast to this, it should be noted that algebraic circuits and rank-1 constraint systems
 5060 are not general enough to express arbitrary computation, but bounded computation only. As a
 5061 consequence, it is not possible to implement unbounded loops, or loops with bounds that are
 5062 unknown at compile time in those models. This can be easily seen since circuits are acyclic
 5063 by definition, and implementing an unbounded loop as an acyclic graph requires a circuits of
 5064 unbounded size.

5065 However, circuits are general enough to express bounded loops, where the upper bound on
 5066 its execution is known at compile time. Those loop can be implemented in circuits by enrolling
 5067 the loop.

something
missing
here?

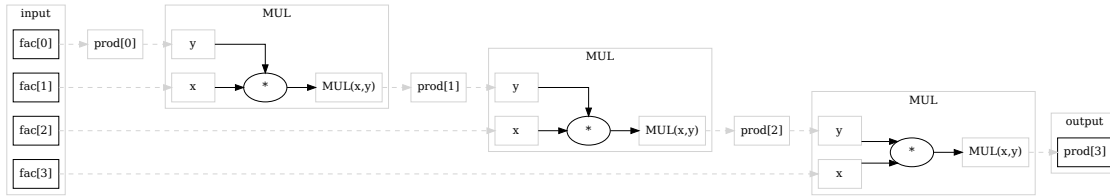
As a consequence, any programming language that compiles to algebraic circuits can only provide loop structures where the bound is a constant known at compile time. This implies that loops cannot depend on execution inputs, but on compile time parameters only.

Example 133. To give an intuition of how a real-world circuit compiler might transform any high-level description of a bounded `for` loop into a circuit, consider the following PAPER code:

```
statement FOR_LOOP {F:F_p, N: unsigned = 4} {
  fn main(fac : F[N]) -> F {
    let prod[N] : F ;
    prod[0] <== fac[0] ;
    for unsigned i in 1..N do [{
      prod[i] <== MUL(fac[i], prod[i-1]) ;
    }]
    return prod[N] ;
  }
}
```

Note that, in a program like this, the loop counter `i` has no expression in the derived circuit. It is pure syntactic **sugar**, telling the compiler how to unroll the loop.

Brain compiling this code into a circuit, we first draw box nodes for all input and output variables, noting that the loop counter is not represented in the circuit. Since all variables are of `field` type, we don't have to compile any type constraints. Then we evaluate the assignments to the output variables by unrolling the loop into 3 individual assignment operators. We get:



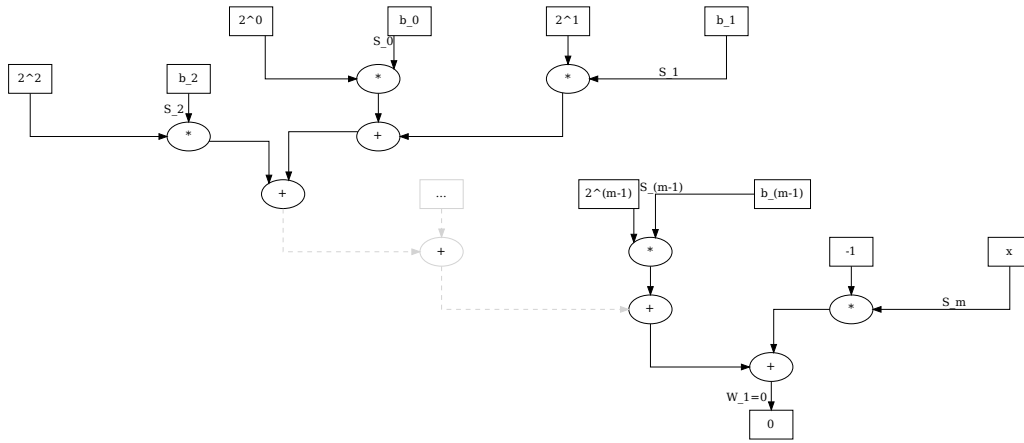
7.2.3 Binary Field Representations

In applications, is often necessary to enforce a binary representation of elements from the `field` type. To derive an appropriate circuit over a prime field \mathbb{F}_p , let $m = |p|_2$ be the smallest number of bits necessary to represent the prime modulus p . Then a bitstring $(b_0, \dots, b_{m-1}) \in \{0, 1\}^m$ is a binary representation of a field element $x \in \mathbb{F}_p$, if and only if

$$x = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_{m-1} \cdot 2^{m-1}$$

In this expression, addition and exponentiation is considered to be executed in \mathbb{F}_p , which is well defined since all terms 2^j for $0 \leq j < m$ are elements of \mathbb{F}_p . Note, however, that in contrast to the binary representation of unsigned integers $n \in \mathbb{N}$, this representation is not unique in general, since the modular p equivalence class might contain more than one binary representative.

Considering that the underlying prime field is fixed and the most significant bit of the prime modulus is m , the following circuit flattens equation XXX, assuming all inputs b_1, \dots, b_m are of boolean type.



5099

Applying the general transformation rule to compute the associated rank-1 constraint systems, we see that we actually only need a single constraint to enforce some binary representation of any field element. We get

$$(S_0 \cdot 2^0 + S_1 \cdot 2^1 + \dots + S_{m-1} \cdot 2^{m-1} - S_m) \cdot 1 = 0$$

5100 Given an array `BOOL[N]` of N boolean constraint field elements and another field element x ,
 5101 the circuit enforces `BOOL[N]` to be one of the binary representations of x . If `BOOL[N]` is not
 5102 a binary representation of x , no valid assignment and hence no solution to the associated R1CS
 5103 can exist.

5104 *Example 134.* Consider the prime field \mathbb{F}_{13} . To compute binary representations of elements
 5105 from that field, we start with the binary representation of the prime modulus 13, which is $|13|_2 =$
 5106 $(1, 0, 1, 1)$ since $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$. So $m = 4$ and we need up to 4 bits to represent
 5107 any element $x \in \mathbb{F}_{13}$.

To see that binary representations are not unique in general, consider the element $2 \in \mathbb{F}_{13}$. It has the binary representations $|2|_2 = (0, 1, 0, 0)$ and $|2|_2 = (1, 1, 1, 1)$, since in \mathbb{F}_{13} we have

$$2 = \begin{cases} 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 \\ 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \end{cases}$$

5108 This is because the unsigned integers 2 and 15 are both in the modular 13 remainder class of 2
 5109 and hence are both representatives of 2 in \mathbb{F}_{13} .

To see how circuit XXX works, we want to enforce the binary representation of $7 \in \mathbb{F}_{13}$. Since $m = 4$ we have to enforce a 4-bit representation for 7, which is $(1, 1, 1, 0)$, since $7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$. A valid circuit assignment is therefore given by $(S_0, S_1, S_2, S_3, S_4) = (1, 1, 1, 0, 7)$ and, indeed, the assignment satisfies the required 5 constraints including the 4 boolean constraints for S_0, \dots, S_3 :

$$\begin{aligned} 1 \cdot (1 - 1) &= 0 & // \text{boolean constraints} \\ 1 \cdot (1 - 1) &= 0 \\ 1 \cdot (1 - 1) &= 0 \\ 0 \cdot (1 - 0) &= 0 \\ (1 + 2 + 4 + 0 - 7) \cdot 1 &= 0 & // \text{binary rep. constraint} \end{aligned}$$

 add refer-
ence

7.2.4 Cryptographic Primitives

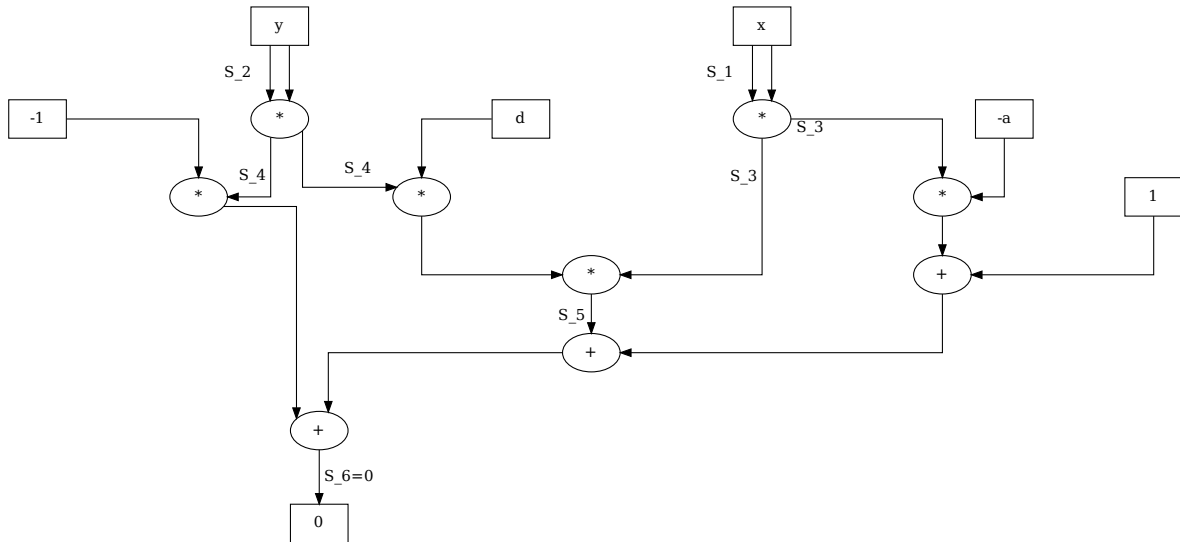
In applications, it is often required to do cryptography in a circuit. To do this, basic cryptographic primitives like hash functions or elliptic curve cryptography needs to be implemented as circuits. In this section, we give a few basic examples of how to implement such primitives.

Twisted Edwards curves

Implementing elliptic curve cryptography in circuits means to implement the field equation as well as the algebraic operations of an elliptic curve as circuits. To do this efficiently, the curve must be defined over the same base field as the field that is used in the circuit.

For efficiency reasons, it is advantageous to choose an elliptic curve such that that all required constraints and operations can be implement with as few gates as possible. Twisted Edwards curves are particularly useful for that matter, since their addition law is particularly simple and the same equation can be used for all curve points including the point at infinity. This simplifies the circuit a lot.

Twisted Edwards curves constraints As we have seen in XXX, a twisted Edwards curve over a finite field F is defined as the set of all pairs of points $(x, y) \in \mathbb{F} \times \mathbb{F}$ such that x and y satisfy the equation $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$. As we have seen in example XXX, we can transform this equation into the following circuit:



The circuit enforces the two inputs of `field` type to satisfy the twisted Edwards curve equation and, as we know from example XXX, the associated rank-1 constraint system is given by:

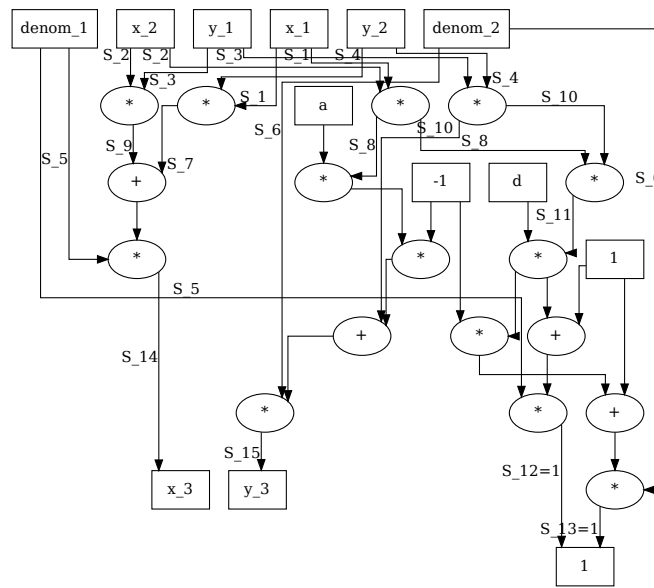
$$\begin{aligned}
 S_1 \cdot S_1 &= S_3 \\
 S_2 \cdot S_2 &= S_4 \\
 (S_4 \cdot 8) \cdot S_3 &= S_5 \\
 (12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 &= 0
 \end{aligned}$$

Exercise 52. Write the circuit and associated rank-1 constraint system for a Weierstraß curve of a given field \mathbb{F} .

Twisted Edwards curve addition As we have seen in XXX, a major advantage of twisted Edwards curves is the existence of an addition law that contains no branching and is valid for all curve points. Moreover the neutral element is not “at infinity” but the actual curve point $(0, 1)$. In fact given two points (x_1, y_1) and (x_2, y_2) on a twisted Edwards curve their sum is defined as

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

5132 We can use the division circuit from XXX to flatten this equation into an algebraic circuit.
 5133 Inputs to the circuit are then not only the two curve points (x_1, y_1) and (x_2, y_2) , but also the two
 5134 denominators $denum_1 = 1 + d \cdot x_1 x_2 y_1 y_2$ as well as $denum_2 = 1 - d \cdot x_1 x_2 y_1 y_2$, which any prover
 5135 needs to compute outside of the circuit. We get



5136

Using the general technique from XXX to derive the associated rank-1 constraint system, we get the following result:

$$\begin{aligned} S_1 \cdot S_4 &= S_7 \\ S_1 \cdot S_2 &= S_8 \\ S_2 \cdot S_3 &= S_9 \\ S_3 \cdot S_4 &= S_{10} \\ S_8 \cdot S_{10} &= S_{11} \\ S_5 \cdot (1 + d \cdot S_{11}) &= 1 \\ S_6 \cdot (1 - d \cdot S_{11}) &= 1 \\ S_5 \cdot (S_9 + S_7) &= S_{14} \\ S_6 \cdot (S_{10} - a \cdot S_8) &= S_{15} \end{aligned}$$

5137 *Exercise 53.* Let \mathbb{F} be a field. Define a circuit that enforces field inversion for a point of a
 5138 twisted Edwards curve over \mathbb{F} .

Chapter 8

Zero Knowledge Protocols

A so called **zero-knowledge protocol** is a set of mathematical rules by which one party usually called **the prover** can convince another party usually called **the verifier** that a given statement is true, while not revealing any additional information apart from the truth of the statement.

As we have seen in chapter XXX, given some language L and instance I the knowledge claim “there is a witness W such that, $(I;W)$ is a word in L ” is constructively provable by providing W to the verifier. However, the challenge for a zero-knowledge protocol is to prove knowledge of a witness without revealing any information beyond its bare existence.

In this chapter, we will look at various systems that exists to solve this task. We start with an introduction to the basic concepts and terminology in zero knowledge proofing systems and then introduce the so called Groth_16 protocol as one of the most efficient systems. We will update the book with new inventions, in future versions of this book.

add reference

8.1 Proof Systems

From an abstract point of view, a proof system is a set of rules which models the generation and exchange of messages between two parties: a prover and a verifier. Its task is to ascertain whether a given string belongs to a formal language or not.

Proof systems are often classified by certain trust assumptions and the computational capabilities of both parties. In it most general form, the prover usually possesses unlimited computational resources but cannot be trusted, while the verifier has bounded computation power but is assumed to be honest.

Proofing the membership statement for some string is then executed by the generation of certain messages that are sent between prover and verifier until the verifier is convinced that the string is an element of the language in consideration.

To be more specific, let Σ be an alphabet and L a formal language defined over Σ . Then a **proof system** for language L is a pair of probabilistic interactive algorithms (P,V) , where P is called the **prover** and V is called the **verifier**.

Both algorithms are able to send messages to one another and each algorithm has its own state, some shared initial state and access to the messages. The verifier is bounded to a number of steps which is polynomial in the size of the shared initial state, after which it stops and output either `accept` or `reject` indicating that it accepts or rejects a given string to be in L . In contrast, there are bounds on the computational power of the prover.

After the execution of the verifier algorithm stops the following conditions are required to hold:

- (Completeness) If the tuple $x \in \Sigma^*$ is a word in language L and both prover and verifier follow the protocol, the verifier outputs `accept`.

- (Soundness) If the tuple $x \in \Sigma^*$ is not a word in language L and the verifier follows the protocol, the verifier outputs `reject`, except with some small probability.

In addition a proof system is called **zero knowledge**, if the verifier learns nothing about x other than $x \in L$.

The previous definition of proof systems is very general and many sub-classes of proofing systems are known in the field. The type of languages any proof system can support, crucially depends on the abilities of the verifier, for example to make random choices, or not, or on the nature and number of the messages that can be exchanged. If the system only requires to send a single message from the prover to the verifier, the proof system is called **non-interactive**, because no interaction other than sending the actual proof is required. In contrast any other proof system is called **interactive**.

A proof system is usually called **succinct**, if the size of the proof is shorter than the witness necessary to generate the proof. Moreover, a proof system is called **computationally sound**, if soundness only holds under the assumption that the computational capabilities of the prover are polynomial bound. To distinguish general proofs from computationally sound proofs, the latter are often called **arguments**. zero-knowledge, succinct, non-interactive arguments of knowledge claims are often called **zk-SNARKs**.

Example 135 (Constructive Proofs for Algebraic Circuits). To formalize our previous notion of constructive proof for algebraic circuits, let \mathbb{F} be a finite field and $C(\mathbb{F})$ an algebraic circuit over \mathbb{F} with associated language $L_{C(\mathbb{F})}$. A non-interactive proof system for $L_{C(\mathbb{F})}$ is given by the following two algorithms:

Given some instance I , the prover algorithm P uses its unlimited computational power to compute a witness W such that, the pair $(I; W)$ is a valid assignment to $C(\mathbb{F})$, whenever the circuit is satisfiable for I . The prover then sends the constructive proof $(I; W)$ to the verifier.

On receiving a message $(I; W)$ the verifier algorithm V assigns the constructive proof $(I; W)$ to circuit $C(\mathbb{F})$ and decides if the assignment is valid, by executing all gates in the circuit. The runtime is polynomial in the number of gates. If the assignment is valid the verifier returns `accepts`, if not it returns `reject`.

To see that this proof system has the completeness and soundness property, let $C(\mathbb{F})$ be a circuit of the field \mathbb{F} and I an instance. The circuit may or may not have a witness W such that, $(I; W)$ is a valid assignment to $C(\mathbb{F})$.

If no W exists, I is not part of any word in $L_{C(\mathbb{F})}$ and there is no way for P to generate a valid assignment. It follows that the verifier will not accept any claimed proof sent by P , which implies that the system has **soundness**.

If on the other hand W exists and P is honest, P can use its unlimited computational power to compute W and send $(I; W)$ to V , which V will accept in polynomial time. This implies that the system has **completeness**.

The system is non-interactive because the prover only sends a single message to the verifier, which contains the proof itself and since in this simple system the witness itself is the proof, the proof system is **not** succinct.

8.2 The “Groth16” Protocol

In chapter XXX we have introduced algebraic circuits, their associated rank-1 constraints systems and their induced quadratic arithmetic programs. These models define formal languages

add reference

and associated membership as well as knowledge claim can be constructively proofed by executing the circuit in order to compute a solution to its associated R1CS. The solution can then be transformed into a polynomial such that, the polynomial is divisible by another polynomial if and only if the solution is correct.

In [XXX] Jens Groth provides a method that can transform those proofs into zero-knowledge succinct non interactive arguments of knowledge. Assuming that pairing groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, b)$ are given, the arguments are of constant size and consist of 2 elements from \mathbb{G}_1 and a single element from \mathbb{G}_2 , regardless of the size of the witness. They are zero-knowledge in the sense, that the verifier learns nothing about the witness, besides the fact that the instance, witness pair is a proper word in the language of the problem.

Verification is non interactive and needs to compute a number of exponentiations proportional to the size of the instance, together with 3 group pairings in order to check a single equation.

The generated argument has perfect completeness, perfect zero-knowledge and soundness in the generic bilinear group model, assuming that a trusted third party exists, that executes a preprocessing phase to generate a common reference string and a simulation trapdoor. This party must be trusted to delete the simulation trapdoor, since everyone in possession of it can simulate proofs.

To be more precise let R be a rank-1 constraints system defined over some finite field \mathbb{F}_r . Then **Groth_16 parameters** for R are given by the set

$$\text{Groth_16} - \text{Param}(R) = (r, \mathbb{G}_1, \mathbb{G}_2, e(\cdot, \cdot), g_1, g_2) \quad (8.1)$$

where \mathbb{G}_1 and \mathbb{G}_2 are finite cyclic groups of order r , g_1 is a generator of \mathbb{G}_1 , g_2 is a generator of \mathbb{G}_2 and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a non-degenerate, bilinear pairing for some target group \mathbb{G}_T . In applications the parameter set is usually agreed on in advance.

Given some Groth_16 parameters a **Groth_16 protocol** is then a quadruple of probabilistic polynomial algorithms (SETUP, PROVE, VFY, SIM) such that

- (Setup-Phase): $(CRS, \tau) \leftarrow \text{Setup}(R)$: Algorithm Setup takes the R1CS R as input and computes a common reference string CRS and a simulation trapdoor τ .
- (Prover-Phase): $\pi \leftarrow \text{Prove}(R, CRS, I, W)$: Given a constructive proof (I, W) for R , algorithm Prove takes the R1CS R , the common reference string CRS and the constructive proof (I, W) as input and computes an zk-SNARK π .
- Verify: $\{\text{accept}, \text{reject}\} \leftarrow \text{Vfy}(R, CRS, I, \pi)$: Algorithm Vfy takes the R1CS R , the common reference string CRS , the instance I and the zk-SNARK π as input and returns `reject` or `accept`.
- $\pi \leftarrow \text{Sim}(R, \tau, CRS, I)$: Algorithm Sim takes the R1CS R , the common reference string CRS , the simulation trapdoor τ and the instance I as input and returns a zk-SNARK π .

We will explain those algorithms together with examples in detail in the appropriate paragraphs of this section.

Assuming a trusted third party for the setup, the protocol is then able to compute a zk-SNARK from a constructive proof for R , assuming that r is sufficiently large and in particular larger then the number of constraints in the associated R1CS.

Example 136 (The 3-Factorization Problem). Consider the 3-factorization problem from XXX and its associated algebraic circuit and rank-1 constraints system from XXX. In this example,

we want to agree on a parameter set $(R, r, \mathbb{G}_1, \mathbb{G}_2, e(\cdot, \cdot), g_1, g_2)$ in order to use the Groth_16 protocol for our 3-factorization problem.

To find proper parameters, first observe that the circuit XXX as well as its associated R1CS $R_{3, \text{fac_zk}}$ XXX and the derived QAP XXX are defined over the field \mathbb{F}_{13} . We therefore have $r = 13$ and need pairing groups \mathbb{G}_1 and \mathbb{G}_2 of order 13.

From XXX we know, that the moon-math curve BLS6 6 has two subgroups $\mathbb{G}_1[13]$ and $\mathbb{G}_2[13]$, that are both of order 13. The associated Weil pairing b XXX is a proper bilinear map. We therefore choose those groups and the Weil pairing together with the generators $g_1 = (13, 15)$ and $g_2 = (7v^2, 16v^3)$ of $\mathbb{G}_1[13]$ and $\mathbb{G}_2[13]$, as parameter

$$\text{Groth_16} - \text{Param}(R_{3, \text{fac_zk}}) = (r, \mathbb{G}_1[13], \mathbb{G}_2[13], e(\cdot, \cdot), (13, 15), (7v^2, 16v^3))$$

It should be noted that our choice is not unique. Every pair of finite cyclic groups of order 13 that has a proper bilinear pairing qualifies as a Groth_16 parameter set. The situation is similar to real world applications, where SNARKS with equivalent behavior are defined over different curves, used in different applications.

The Setup Phase To generate zk-SNARKs from constructive knowledge proofs in the Groth16 protocol, a preprocessing phase is required that has to be executed a single time for every rank-1 constraints system and any associated quadratic arithmetic program. The outcome of this phase is a common reference string, that proofer and verifier need to generate and verify the zk-SNARK. In addition a simulation trapdoor is produced that can be used to simulate proofs.

To be more precise, let L be a language defined by some rank-1 constraints system R such that, a constructive proof of knowledge for an instance (I_1, \dots, I_n) in L consists of a witness (W_1, \dots, W_m) . Let $QAP(R) = \{T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m}\}$ be a quadratic arithmetic program associated to R and $\{\mathbb{G}_1, \mathbb{G}_2, e(\cdot, \cdot), g_1, g_2, \mathbb{F}_r\}$ be the set of Groth_16 parameters.

The setup phase then samples 5 random, **inverible** elements $\alpha, \beta, \gamma, \delta$ and s from the scalar field \mathbb{F}_r of the protocol and outputs the **simulation trapdoor**

$$\tau = (\alpha, \beta, \gamma, \delta, s) \quad (8.2)$$

In addition the setup phase uses those 5 random elements together with the two generators g_1 and g_2 and the quadratic arithmetic program, to generate a **common reference string** $CRS_{QAP} = (CRS_{\mathbb{G}_1}, CRS_{\mathbb{G}_2})$ of language L :

$$CRS_{\mathbb{G}_1} = \left\{ g_1^\alpha, g_1^\beta, g_1^\delta, \left(g_1^{s^j}, \dots \right)_{j=0}^{\deg(T)-1}, \left(g_1^{\frac{\beta \cdot A_j(s) + \alpha \cdot B_j(s) + C_j(s)}{\gamma}}, \dots \right)_{j=0}^n, \left(g_1^{\frac{\beta \cdot A_{j+n}(s) + \alpha \cdot B_{j+n}(s) + C_{j+n}(s)}{\delta}}, \dots \right)_{j=1}^m, \left(g_1^{\frac{s^j \cdot T(s)}{\delta}}, \dots \right)_{j=0}^{\deg(T)-2} \right\}$$

$$CRS_{\mathbb{G}_2} = \left\{ g_2^\beta, g_2^\gamma, g_2^\delta, \left(g_2^{s^j}, \dots \right)_{j=0}^{\deg(T)-1} \right\}$$

Common reference strings depend on the simulation trapdoor and are therefor not unique to the problem. Any language can have more than one common reference string. The size of a common reference string is linear in the size of the instance and the size witness.

If a simulation trapdoor $\tau = (\alpha, \beta, \gamma, \delta, s)$ is given, we call the element s a **secret evaluation point** of the protocol, because if \mathbb{F}_r is the scalar field of the finite cyclic groups \mathbb{G}_1 and \mathbb{G}_2 then

a key feature of any common reference string is, that it provides data to compute the evaluation of any polynomial $P \in \mathbb{F}_r[x]$ of degree $\deg(P) < \deg(T)$ at the point s in the exponent of the generator g_1 or g_2 , without knowing s .

To be more precise, let s be the secret evaluation point and $P(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + \dots + a_k \cdot x^k$ a polynomial of degree $k < \deg(T)$ with coefficients in \mathbb{F}_r . Then we can compute $g_1^{P(s)}$ without knowing what the actual value of s is:

$$\begin{aligned} g_1^{P(s)} &= g_1^{a_0 \cdot s^0 + a_1 \cdot s^1 + \dots + a_k \cdot s^k} \\ &= g_1^{a_0 \cdot s^0} \cdot g_1^{a_1 \cdot s^1} \cdot \dots \cdot g_1^{a_k \cdot s^k} \\ &= (g_1^{s^0})^{a_0} \cdot (g_1^{s^1})^{a_1} \cdot \dots \cdot (g_1^{s^k})^{a_k} \end{aligned}$$

In this expression all the group points $g_1^{s^j}$ are part of the common reference string and hence can be used to compute the result. The same holds true for the evaluation of $g_2^{P(s)}$ since the \mathbb{G}_2 part of the common reference string contains the points $g_2^{s^j}$.

In real world applications, the simulation trapdoor is often called **toxic waste** of the setup-phase, while a common reference string is also called a pair of **proofer and verifier key**.

In order to make the protocol secure the setup needs to be executed in a way such that, it is guaranteed that the simulation trapdoor is deleted. Anyone in possession of it can generate arguments without knowledge of a constructive proof. The most simple approach to achieve deletion of the toxic waste is by a so called **trusted third party**, where the trust assumption is, that that the party generates the common reference string precisely as defined and deletes the simulation backdoor afterwards.

However, as trusted third parties are not easy to find in real world application more sophisticated protocols exists that execute the setup phase as a multi party computation, where the proper execution can be publicly verified and the simulation trapdoor is deleted if at least one participants deletes their individual contribution to the randomness. Each participant only possesses a fraction of the simulation trapdoor and the toxic waste can only be recovered if all participants collude and share their fraction.

Example 137 (The 3-factorization Problem). To see how the setup phase of a Groth_16 zk-SNARK can be computed, consider the 3-factorization problem from XXX and the parameters from XXX. As we have seen in XXX an associated quadratic arithmetic program is given by

$$\begin{aligned} QAP(R_{3, fac_zk}) &= \{x^2 + x + 9, \\ &\quad \{0, 0, 6x + 10, 0, 0, 7x + 4\}, \{0, 0, 0, 6x + 10, 7x + 4, 0\}, \{0, 7x + 4, 0, 0, 0, 6x + 10\}\} \end{aligned}$$

To transform this QAP into a common reference string, we choose the following field elements $\alpha = 6$, $\beta = 5$, $\gamma = 4$, $\delta = 3$, $s = 2$ from \mathbb{F}_{13} . In real world applications it is important to sample those values randomly from the scalar field, but in our approach, we choose those non random values to make them more memorable, which helps in pen and paper computations. Our simulation trapdoor is then given by

$$\tau = (6, 5, 4, 3, 2)$$

and we keep this secret in order to simulate proofs later on. We are careful though to hide τ from anyone who hasn't read this book. From those values we then instantiate the common

add reference

add reference

add reference

5307 reference string XXX. Since our groups are subgroups of the BLS6_6 elliptic curve, we use
 5308 scalar product notation instead of exponentiation.

To compute the \mathbb{G}_1 part of the common reference string we use the logarithmic order of the group \mathbb{G}_1 XXX and the generator $g_1 = (13, 15)$ as well as the values from the simulation backdoor. Since $\deg(T) = 2$, we get:

$$\begin{aligned} [\alpha]_{g_1} &= [6](13, 15) = (27, 34) \\ [\beta]_{g_1} &= [5](13, 15) = (26, 34) \\ [\delta]_{g_1} &= [3](13, 15) = (38, 15) \end{aligned}$$

To compute the rest of the \mathbb{G}_1 part of the common reference string, we expand the indexed tuples and insert the secret random elements from the simulation backdoor. We get

$$\begin{aligned} \left([s^j]_{g_1}, \dots \right)_{j=0}^1 &= \left([2^0](13, 15), [2^1](13, 15) \right) \\ &= \left((13, 15), (33, 34) \right) \\ \left(\left[\frac{\beta A_j(s) + \alpha B_j(s) + C_j(s)}{\gamma} \right]_{g_1}, \dots \right)_{j=0}^1 &= \left(\left[\frac{5A_0(2) + 6B_0(2) + C_0(2)}{4} \right](13, 15), \right. \\ &\quad \left. \left[\frac{5A_1(2) + 6B_1(2) + C_1(2)}{4} \right](13, 15) \right) \\ \left(\left[\frac{\beta A_{j+n}(s) + \alpha B_{j+n}(s) + C_{j+n}(s)}{\delta} \right]_{g_1}, \dots \right)_{j=1}^4 &= \left(\left[\frac{5A_2(2) + 6B_2(2) + C_2(2)}{3} \right](13, 15), \right. \\ &\quad \left[\frac{5A_3(2) + 6B_3(2) + C_3(2)}{3} \right](13, 15), \\ &\quad \left[\frac{5A_4(2) + 6B_4(2) + C_4(2)}{3} \right](13, 15), \\ &\quad \left. \left[\frac{5A_5(2) + 6B_5(2) + C_6(2)}{3} \right](13, 15) \right) \\ \left(\left[\frac{s^j \cdot T(s)}{\delta} \right]_{g_1} \right)_{j=0}^0 &= \left(\left[\frac{2^0 \cdot T(2)}{3} \right](13, 15) \right) \end{aligned}$$

To compute the curve points on the right side of these expressions we need the polynomials from the associated quadratic arithmetic program and evaluates them on the secret point $s = 2$.

Since $4^{-1} = 10$ and $3^{-1} = 9$ in \mathbb{F}_{13} , we get

$$\left[\frac{5A_0(2) + 6B_0(2) + C_0(2)}{4} \right](13, 15) = [(5 \cdot 0 + 6 \cdot 0 + 0) \cdot 10](13, 15) = [0](13, 15) =$$

\emptyset

$$\left[\frac{5A_1(2) + 6B_1(2) + C_1(2)}{4} \right](13, 15) = [(5 \cdot 0 + 6 \cdot 0 + (7 \cdot 2 + 4)) \cdot 10](13, 15) = [11](13, 15) =$$

$(33, 9)$

$$\left[\frac{5A_2(2) + 6B_2(2) + C_2(2)}{3} \right](13, 15) = [(5 \cdot (6 \cdot 2 + 10) + 6 \cdot 0 + 0) \cdot 9](13, 15) = [2](13, 15) =$$

$(33, 34)$

$$\left[\frac{5A_3(2) + 6B_3(2) + C_3(2)}{3} \right](13, 15) = [(5 \cdot 0 + 6 \cdot (6 \cdot 2 + 10) + 0) \cdot 9](13, 15) = [5](13, 15) =$$

$(26, 34)$

$$\left[\frac{5A_4(2) + 6B_4(2) + C_4(2)}{3} \right](13, 15) = [(5 \cdot 0 + 6 \cdot (7 \cdot 2 + 4) + 0) \cdot 9](13, 15) = [10](13, 15) =$$

$(38, 28)$

$$\left[\frac{5A_5(2) + 6B_5(2) + C_5(2)}{3} \right](13, 15) = [(5 \cdot (7 \cdot 2 + 4) + 6 \cdot 0 + 0) \cdot 9](13, 15) = [4](13, 15) =$$

$(35, 28)$

$$\left[\frac{2^0 \cdot T(2)}{3} \right](13, 15) = [1 \cdot (2^2 + 2 + 9) \cdot 9](13, 15) = [5](13, 15) =$$

$(26, 34)$

Putting all those values together we see that the \mathbb{G}_1 part of the common reference string is given by the following set of 12 points from the BLS6_6 13-torsion group \mathbb{G}_1 :

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{l} (27, 34), (26, 34), (38, 15), \left((13, 15), (33, 34) \right), \left(\emptyset, (33, 9) \right) \\ \left((33, 34), (26, 34), (38, 28), (35, 28) \right), \left((26, 34) \right) \end{array} \right\}$$

To compute the \mathbb{G}_2 part of the common reference string we use the logarithmic order of the group \mathbb{G}_2 XXX and the generator $g_2 = (7v^2, 16v^3)$ as well as the values from the simulation backdoor. Since $\deg(T) = 2$, we get:

$$[\beta]g_2 = [5](7v^2, 16v^3) = (16v^2, 28v^3)$$

$$[\gamma]g_2 = [4](7v^2, 16v^3) = (37v^2, 27v^3)$$

$$[\delta]g_2 = [3](7v^2, 16v^3) = (42v^2, 16v^3)$$

To compute the rest of the \mathbb{G}_2 part of the common reference string, we expand the indexed tuple and insert the secret random elements from the simulation backdoor. We get

$$\begin{aligned} \left([s^j]g_2, \dots \right)_{j=0}^1 &= ([2^0](7v^2, 16v^3), [2^1](7v^2, 16v^3)) \\ &= ((7v^2, 16v^3), (10v^2, 28v^3)) \end{aligned}$$

Putting all those values together we see that the \mathbb{G}_2 part of the common reference string is given by the following set of 5 points from the BLS6_6 13-torsion group \mathbb{G}_2 :

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

add reference

Given the simulation trapdoor τ and the quadratic arithmetic program XXX, the associated common reference string of the 3-factorization problem is given by

add reference

$$\begin{aligned} CRS_{\mathbb{G}_1} &= \left\{ (27, 34), (26, 34), (38, 15), \left((13, 15), (33, 34) \right), \left(\emptyset, (33, 9) \right) \right\} \\ &\quad \left\{ \left((33, 34), (26, 34), (38, 28), (35, 28) \right), \left((26, 34) \right) \right\} \\ CRS_{\mathbb{G}_2} &= \left\{ (16v^2, 28v^3), (37v^2, 27v^3), (42v^2, 16v^3), (7v^2, 16v^3), (10v^2, 28v^3) \right\} \end{aligned}$$

5309 We then publish this data to everyone who wants to participate in the zk-SNARK generation or
5310 verification of the 3-factorization problem.

5311 To understand how this common reference string can be used, to evaluate polynomials at
5312 the secret evaluation point in the exponent of a generator, let's assume that we have deleted the
5313 simulation trapdoor. In that case we have no way to know the secret evaluation point anymore
5314 and hence can not evaluate polynomials at that point. However, we can evaluate polynomials
5315 of degree smaller than the degree of the target polynomial in the exponent of both generators at
5316 that point.

To see that consider for example the polynomials $A_2(x) = 6x + 10$ and $A_5(x) = 7x + 4$ from the QAP of this problem. To evaluate these polynomials in the exponent of g_1 and g_2 at the secret point s , without knowing the value of s (which is 2), we can use the common reference string and equation XXX. Using the scalar product notation, instead of exponentiation, we get

add reference

$$\begin{aligned} [A_2(s)]_{g_1} &= [6 \cdot s^1 + 10 \cdot s^0]_{g_1} \\ &= [6](33, 34) + [10](13, 15) & \# [s^0]_{g_1} = (13, 15), [s^1]_{g_1} = (33, 34) \\ &= [6 \cdot 2](13, 15) + [10](13, 15) = [9](13, 15) & \# \text{logarithmic order on } \mathbb{G}_1 \\ &= (35, 15) \\ [A_5(s)]_{g_1} &= [7 \cdot s^1 + 4 \cdot s^0]_{g_1} \\ &= [7](33, 34) + [4](13, 15) \\ &= [7 \cdot 2](13, 15) + [4](13, 15) = [5](13, 15) \\ &= (26, 34) \end{aligned}$$

Indeed we are able to evaluate the polynomials in the exponent at a secret evaluation point because that point is encrypted in the curve point $(33, 34)$ and its secrecy is protected by the discrete logarithm assumption. Of course in our computation we recovered the secret point $s = 2$, but that was only possible, because we have a logarithmic ordering of the group to simplify our pen and paper computations. Such an order is infeasible to compute in cryptographically secure curves. We can do the same computation on \mathbb{G}_2 and get

$$\begin{aligned} [A_2(s)]_{g_2} &= [6 \cdot s^1 + 10 \cdot s^0]_{g_2} \\ &= [6](10v^2, 28v^3) + [10](7v^2, 16v^3) \\ &= [6 \cdot 2](7v^2, 16v^3) + [10](7v^2, 16v^3) = [9](7v^2, 16v^3) \\ &= (37v^2, 16v^3) \\ [A_5(s)]_{g_2} &= [7 \cdot s^1 + 4 \cdot s^0]_{g_2} \\ &= [7](10v^2, 28v^3) + [4](7v^2, 16v^3) \\ &= [7 \cdot 2](7v^2, 16v^3) + [4](7v^2, 16v^3) = [5](7v^2, 16v^3) \\ &= (16v^2, 28v^3) \end{aligned}$$

5317 Except for the target polynomial T all other polynomials of the quadratic arithmetic program
 5318 can be evaluated in the exponent this way.

5319 **The Proofer Phase** Given some rank-1 constraints system R and instance $I = (I_1, \dots, I_n)$, the
 5320 task of the proofer phase is to convince any verifier, that a proofer knows a witness W to instance
 5321 I such that, $(I; W)$ is a word in the language L_R of the system, without revealing anything about
 5322 W .

5323 To achieve this in the Groth_16 protocol, we assume that any proofer has access to the rank-
 5324 1 constraints system of the problem in addition with some algorithm, that tells the proofer how
 5325 to compute constructive proofs for the R1CS. In addition the proofer has access to a common
 5326 reference string and its associated quadratic arithmetic program.

5327 In order to generate a zk-SNARK for this instance, the proofer first computes a valid
 5328 constructive proof as explained in XXX, that is the proofer generates a proper witness $W =$
 5329 (W_1, \dots, W_m) such that, $(I_1, \dots, I_n; W_1, \dots, W_m)$ is a solution to the rank-1 constraints system R .

5330 The proofer then uses the quadratic arithmetic program and computes the polynomial $P_{(I;W)}$
 5331 as explained in XXX. They then divide $P_{(I;W)}$ by the target polynomial T of the quadratic arith-
 5332 metic. Since $P_{(I;W)}$ is constructed from a valid solution to the R1CS we know from XXX that it
 5333 is divisible by T . This implies that polynomial division of P by T generates another polynomial
 5334 $H := P/T$, with $\deg(H) < \deg(T)$.

5335 The proofer then evaluates the polynomial $(H \cdot T)\delta^{-1}$ in the exponent of the generator g_1 at
 5336 the secret point s as explained in XXX. To see how this can be achieved, let

$$H(x) = H_0 \cdot x^0 + H_1 \cdot x^1 + \dots + H_k \cdot x^k \quad (8.3)$$

be the quotient polynomial P/T . To evaluate $H \cdot T$ at s in the exponent of g_1 , the proofer uses
 the common reference string and computes

$$g_1^{\frac{H(s) \cdot T(s)}{\delta}} = \left(g_1^{\frac{s^0 \cdot T(s)}{\delta}}\right)^{H_0} \cdot \left(g_1^{\frac{s^1 \cdot T(s)}{\delta}}\right)^{H_1} \cdots \left(g_1^{\frac{s^k \cdot T(s)}{\delta}}\right)^{H_k}$$

After this has been done, the proofer samples two random field elements $r, t \in \mathbb{F}_r$ and uses the
 common reference string, the instance variables I_1, \dots, I_n and the witness variables W_1, \dots, W_m
 to compute the following curve points

$$\begin{aligned} g_1^W &= \left(g_1^{\frac{\beta \cdot A_{1+n}(s) + \alpha \cdot B_{1+n}(s) + C_{1+n}(s)}{\delta}}\right)^{W_1} \cdots \left(g_1^{\frac{\beta \cdot A_{m+n}(s) + \alpha \cdot B_{m+n}(s) + C_{m+n}(s)}{\delta}}\right)^{W_m} \\ g_1^A &= g_1^\alpha \cdot g_1^{A_0(s)} \cdot \left(g_1^{A_1(s)}\right)^{I_1} \cdots \left(g_1^{A_n(s)}\right)^{I_n} \cdot \left(g_1^{A_{n+1}(s)}\right)^{W_1} \cdots \left(g_1^{A_{n+m}(s)}\right)^{W_m} \cdot \left(g_1^\delta\right)^r \\ g_1^B &= g_1^\beta \cdot g_1^{B_0(s)} \cdot \left(g_1^{B_1(s)}\right)^{I_1} \cdots \left(g_1^{B_n(s)}\right)^{I_n} \cdot \left(g_1^{B_{n+1}(s)}\right)^{W_1} \cdots \left(g_1^{B_{n+m}(s)}\right)^{W_m} \cdot \left(g_1^\delta\right)^t \\ g_2^B &= g_2^\beta \cdot g_2^{B_0(s)} \cdot \left(g_2^{B_1(s)}\right)^{I_1} \cdots \left(g_2^{B_n(s)}\right)^{I_n} \cdot \left(g_2^{B_{n+1}(s)}\right)^{W_1} \cdots \left(g_2^{B_{n+m}(s)}\right)^{W_m} \cdot \left(g_2^\delta\right)^t \\ g_1^C &= g_1^W \cdot g_1^{\frac{H(s) \cdot T(s)}{\delta}} \cdot \left(g_1^A\right)^t \cdot \left(g_1^B\right)^r \cdot \left(g_1^\delta\right)^{-r \cdot t} \end{aligned}$$

5337 In this computation, the group elements $g_1^{A_j(s)}$, $g_1^{B_j(s)}$ and $g_2^{B_j(s)}$ can be derived from the common
 5338 reference string and the quadratic arithmetic program of the problem, as we have seen in XXX.
 5339 In fact those points only have to be computed once and can be published and reused for multiple
 5340 proof generations as they are the same for all instances and witnesses. All other group elements
 5341 are part of the common reference string.

After all these computations have been done, a valid zero-knowledge succinct non-interactive argument of knowledge π in the Groth_16 protocol is given by the following three curve points

$$\pi = (g_1^A, g_1^C, g_2^B) \quad (8.4)$$

As we can see, a Groth_16 zk-SNARK consists of 3 curve points. Two points from \mathbb{G}_1 and 1 point from \mathbb{G}_2 . The argument is specifically designed this way, because in typical applications \mathbb{G}_1 is a torsion group of an elliptic curve over some prime field, while \mathbb{G}_2 is a subgroup of a torsion group over an extension field. Elements from \mathbb{G}_1 therefore need less space to be stored and computations in \mathbb{G}_1 are typically faster than in \mathbb{G}_2 .

Since the witness is encoded in the exponent of a generator of a cryptographically secure elliptic curve, it is hidden from anyone but the prover. Moreover, since any proof is randomized by the occurrence of the random field elements r and t , proofs are not unique for any given witness. This is an important feature, because if all proofs for the same witness would be the same, knowledge of a witness would destroy the zero knowledge property of those proofs.

Example 138 (The 3-factorization Problem). To see how a prover might compute a zk-SNARK, consider the 3-factorization problem from XXX, our protocol parameters from XXX as well as the common reference string from XXX.

Our task is to compute a zk-SNARK for the instance $I_1 = 11$ and its constructive proof $(W_1, W_2, W_3, W_4) = (2, 3, 4, 6)$ as computed in XXX. As we know from XXX the associated polynomial $P_{(I;W)}$ of the quadratic arithmetic program from XXX is given by

$$P_{(I;W)} = x^2 + x + 9$$

and since in this example $P_{(I;W)}$ is identical to the target polynomial $T(x) = x^2 + x + 9$, we know from XXX, that the quotient polynomial $H = P/T$ is the constant degree 0 polynomial

$$H(x) = H_0 \cdot x^0 = 1 \cdot x^0$$

We therefore use $[\frac{s^0 \cdot T(s)}{\delta}]_{g_1} = (26, 34)$ from our common reference string XXX of the 3-factorization problem and compute

$$\begin{aligned} [\frac{H(s) \cdot T(s)}{\delta}]_{g_1} &= [H_0](26, 34) = [1](26, 34) \\ &= (26, 34) \end{aligned}$$

In a next step we have to compute all group elements required for a proper Groth16 zk-SNARK. We start with g_1^W . Using scalar products instead of the exponential notation and \oplus for the group law on the BLS6_6 curve, we have to compute the point

$$\begin{aligned} [W]g_1 &= [W_1]g_1 \frac{\beta \cdot A_2(s) + \alpha \cdot B_2(s) + C_2(s)}{\delta} \oplus [W_2]g_1 \frac{\beta \cdot A_3(s) + \alpha \cdot B_3(s) + C_3(s)}{\delta} \oplus [W_3]g_1 \frac{\beta \cdot A_4(s) + \alpha \cdot B_4(s) + C_4(s)}{\delta} \\ &\quad \oplus [W_4]g_1 \frac{\beta \cdot A_5(s) + \alpha \cdot B_5(s) + C_5(s)}{\delta} \end{aligned}$$

To compute this point, we have to remember that a prover should not be in possession of the simulation trapdoor and hence does not know what α , β , δ and s are. In order to compute this group element, the prover therefore needs the common reference string. Using the logarithmic order from XXX and the witness we get

add reference

add reference

add reference

add reference

add reference

add reference

add reference

add reference

add reference

$$\begin{aligned}
 [W]g_1 &= [2](33, 34) \oplus [3](26, 34) \oplus [4](38, 28) \oplus [6](35, 28) \\
 &= [2 \cdot 2](13, 15) \oplus [3 \cdot 5](13, 15) \oplus [4 \cdot 10](13, 15) \oplus [6 \cdot 4](13, 15) \\
 &= [2 \cdot 2 + 3 \cdot 5 + 4 \cdot 10 + 6 \cdot 4](13, 15) = [5](13, 15) \\
 &= (26, 34)
 \end{aligned}$$

In a next step we compute g_1^A . We sample the random point $r = 11$ from \mathbb{F}_{13} , use scalar products instead of the exponential notation and \oplus for the group law on the BLS6_6 curve. We then have to compute the following expression

$$\begin{aligned}
 [A]g_1 &= [\alpha]g_1 \oplus [A_0(s)]g_1 \oplus [I_1][A_1(s)]g_1 \oplus [W_1][A_2(s)]g_1 \oplus [W_2][A_3(s)]g_1 \\
 &\quad \oplus [W_3][A_4(s)]g_1 \oplus [W_4][A_5(s)]g_1 \oplus [r][\delta]g_1
 \end{aligned}$$

Since we don't know what α , δ and s are we look up $[\alpha]g_1$ and $[\delta]g_1$ from the common reference string and recall from XXX that we can evaluate $[A_j(s)]g_1$ without knowledge of the secret evaluation point s . According to XXX we have $[A_2(s)]g_1 = (35, 15)$, $[A_5(s)]g_1 = (26, 34)$ and $[A_j(s)]g_1 = \mathcal{O}$ for all other indices $0 \leq j \leq 5$. Since \mathcal{O} is the neutral element on \mathbb{G}_1 , we get

$$\begin{aligned}
 [A]g_1 &= (27, 34) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2](35, 15) \oplus [3]\mathcal{O} \oplus [4]\mathcal{O} \oplus [6](26, 34) \oplus [11](38, 15) \\
 &= (27, 34) \oplus [2](35, 15) \oplus [6](26, 34) \oplus [11](38, 15) \\
 &= [6](13, 15) \oplus [2 \cdot 9](13, 15) \oplus [6 \cdot 5](13, 15) \oplus [11 \cdot 3](13, 15) \\
 &= [6 + 2 \cdot 9 + 6 \cdot 5 + 11 \cdot 3](13, 15) = [9](13, 15) \\
 &= (35, 15)
 \end{aligned}$$

add reference

add reference

In order to compute the two curve points $[B]g_1$ and $[B]g_2$, we sample another random element $t = 4$ from \mathbb{F}_{13} . Using the scalar product instead of the exponential notation and \oplus for the group law on the BLS6_6 curve, we have to compute the following expressions

$$\begin{aligned}
 [B]g_1 &= [\beta]g_1 \oplus [B_0(s)]g_1 \oplus [I_1][B_1(s)]g_1 \oplus [W_1][B_2(s)]g_1 \oplus [W_2][B_3(s)]g_1 \\
 &\quad \oplus [W_3][B_4(s)]g_1 \oplus [W_4][B_5(s)]g_1 \oplus [t][\delta]g_1 \\
 [B]g_2 &= [\beta]g_2 \oplus [B_0(s)]g_2 \oplus [I_1][B_1(s)]g_2 \oplus [W_1][B_2(s)]g_2 \oplus [W_2][B_3(s)]g_2 \\
 &\quad \oplus [W_3][B_4(s)]g_2 \oplus [W_4][B_5(s)]g_2 \oplus [t][\delta]g_2
 \end{aligned}$$

Since we don't know what β , δ and s are we look up the associated group elements from the common reference string and recall from XXX that we can evaluate $[B_j(s)]g_1$ without knowledge of the secret evaluation point s . Since $B_3 = A_2$ as well as $B_4 = A_5$, we have $[B_3(s)]g_1 = (35, 15)$, $[B_4(s)]g_1 = (26, 34)$ according to XXX and $[B_j(s)]g_1 = \mathcal{O}$ for all other indices $0 \leq j \leq 5$. Since \mathcal{O} is the neutral element on \mathbb{G}_1 , we get

add reference

add reference

$$\begin{aligned}
 [B]g_1 &= (26, 34) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2]\mathcal{O} \oplus [3](35, 15) \oplus [4](26, 34) \oplus [6]\mathcal{O} \oplus [4](38, 15) \\
 &= (26, 34) \oplus [3](35, 15) \oplus [4](26, 34) \oplus [4](38, 15) \\
 &= [5](13, 15) \oplus [3 \cdot 9](13, 15) \oplus [4 \cdot 5](13, 15) \oplus [4 \cdot 3](13, 15) \\
 &= [5 + 3 \cdot 9 + 4 \cdot 5 + 4 \cdot 3](13, 15) = [12](13, 15) \\
 &= (13, 28)
 \end{aligned}$$

$$\begin{aligned}
 [B]g_2 &= (16v^2, 28v^3) \oplus \mathcal{O} \oplus [11]\mathcal{O} \oplus [2]\mathcal{O} \oplus [3](37v^2, 16v^3) \oplus [4](16v^2, 28v^3) \oplus [6]\mathcal{O} \oplus [4](42v^2, 16v^3) \\
 &= (16v^2, 28v^3) \oplus [3](37v^2, 16v^3) \oplus [4](16v^2, 28v^3) \oplus [4](42v^2, 16v^3) \\
 &= [5](7v^2, 16v^3) \oplus [3 \cdot 9](7v^2, 16v^3) \oplus [4 \cdot 5](7v^2, 16v^3) \oplus [4 \cdot 3](7v^2, 16v^3) \\
 &= [5 + 3 \cdot 9 + 4 \cdot 5 + 4 \cdot 3](7v^2, 16v^3) = [12](7v^2 + 16v^3) \\
 &= (7v^2, 27v^3)
 \end{aligned}$$

In a last step we can combine the previous computations, to compute the point $[C]g_1$ in the group \mathbb{G}_1 . we get

$$\begin{aligned}
 [C]g_1 &= [W]g_1 \oplus \left[\frac{H(s) \cdot T(s)}{\delta} \right]g_1 \oplus [t][A]g_1 \oplus [r][B]g_1 \oplus [-r \cdot t][\delta]g_1 \\
 &= (26, 34) \oplus (26, 34) \oplus [4](35, 15) \oplus [11](13, 28) \oplus [-11 \cdot 4](38, 15) \\
 &= [5](13, 15) \oplus [5](13, 15) \oplus [4 \cdot 9](13, 15) \oplus [11 \cdot 12](13, 15) \oplus [-11 \cdot 4 \cdot 3](13, 15) \\
 &= [5 + 5 + 4 \cdot 9 + 11 \cdot 12 - 11 \cdot 4 \cdot 3](13, 15) = [7](13, 15) \\
 &= (27, 9)
 \end{aligned}$$

Given instance $I_1 = 11$ we can now combine those computation and see that the following 3 curve points are a zk-SNARK for the witness $(W_1, W_2, W_3, W_4) = (2, 3, 4, 6)$:

$$\pi = ((35, 15), (27, 9), (7v^2, 27v^3))$$

5358 We can publish this zk-SNARK or send it to a designated verifier. Note that if we had sam-
 5359 pled different values for r and t , we would have computed a different SNARK for the same
 5360 witness. The SNARK therefore hides the witness perfectly, which means that it is impossible
 5361 to reconstruct the witness from the SNARK.

5362 **The Verification Phase** Given some rank-1 constraints system R , instance $I = (I_1, \dots, I_n)$
 5363 and zk-SNARK π , the task of the verifier phase is to check that π is indeed an argument for
 5364 a constructive proof. Assuming that the simulation trapdoor does not exist anymore and the
 5365 verification checks the proof, the verifier can be convinced, that someone knows a witness
 5366 $W = (W_1, \dots, W_m)$ such that, $(I; W)$ is a word in the language of R .

To achieve this in the Groth16 protocol, we assume that any verifier is able to compute the pairing map $e(\cdot, \cdot)$ efficiently and has access to the common reference string used to produce the SNARK π . In order to verify the SNARK with respect to the instance (I_1, \dots, I_n) , the verifier computes the following curve point:

$$g_1^I = \left(g_1^{\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma}} \right) \cdot \left(g_1^{\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma}} \right)^{I_1} \cdots \left(g_1^{\frac{\beta \cdot A_n(s) + \alpha \cdot B_n(s) + C_n(s)}{\gamma}} \right)^{I_n}$$

5367 With this group element the verifier is then able to verify the SNARK $\pi = (g_1^A, g_1^C, g_2^B)$ by
 5368 checking the following equation using the pairing map:

$$e(g_1^A, g_2^B) = e(g_1^\alpha, g_2^\beta) \cdot e(g_1^I, g_2^\gamma) \cdot e(g_1^C, g_2^\delta) \quad (8.5)$$

5369 If the equation holds true, the SNARK is accepted and if the equation does not hold, the SNARK
 5370 is rejected.

Remark 5. We know from XXX that computing pairings in cryptographically secure pairing groups is computationally expensive. As we can see, in the Groth16 protocol 3 pairings are required to verify the SNARK, because the pairing $e(g_1^\alpha, g_2^\beta)$ is independent of the proof and can be computed once and then stored as an amendment to the verifier key.

add reference

In [GROTH16] the author showed that 2 pairings is the minimal amount of pairings that any protocol with similar properties has to use. This protocol is therefore close to the theoretic minimum. In the same paper the author outlined an adaptation that only uses 2 pairings. However, that reduction comes with the price of much more overhead computation. 3 pairings is therefore a compromise that gives the overall best performance. To date the Groth16 protocol is the most efficient in its class.

Example 139 (The 3-factorization Problem). To see how a verifier might check a zk-SNARK for some given instance I , consider the 3-factorization problem from XXX, our protocol parameters from XXX, the common reference string from XXX as well as the zk-SNARK $\pi = ((35, 15), (27, 9), (7v^2, 27v^3))$, which claims to be an argument of knowledge for a witness for the instance $I_1 = 11$.

add reference

add reference

In order to verify the zk-SNARK for that instance, we first compute the curve point g_1^I . Using scalar products instead of the exponential notation and \oplus for the group law on the BLS6_6 curve, we have to compute the point

add reference

$$[I]_{g_1} = \left[\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma} \right]_{g_1} \oplus [I_1] \left[\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma} \right]_{g_1}$$

To compute this point, we have to remember that a verifier should not be in possession of the simulation trapdoor and hence does not know what α , β , γ and s are. In order to compute this group element, the verifier therefore need the common reference string. Using the logarithmic order from XXX and instance I_1 we get

add reference

$$\begin{aligned} [I]_{g_1} &= \left[\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma} \right]_{g_1} \oplus [I_1] \left[\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma} \right]_{g_1} \\ &= \mathcal{O} \oplus [11](33, 9) \\ &= [11 \cdot 11](13, 15) = [4](13, 15) \\ &= (35, 28) \end{aligned}$$

In a next step, we have to compute all the pairings involved in equation XXX. Using the logarithmic

add reference

rithmic order on \mathbb{G}_1 and \mathbb{G}_2 as well as the bilinearity property of the pairing map we get

$$\begin{aligned}
 e([A]g_1, [B]g_2) &= e((35, 15), (7v^2, 27v^3)) = e([9](13, 15), [12](7v^2, 16v^3)) \\
 &= e((13, 15), (7v^2, 16v^3))^{9 \cdot 12} \\
 &= e((13, 15), (7v^2, 16v^3))^{108} \\
 e([\alpha]g_1, [\beta]g_2) &= e((27, 34), (16v^2, 28v^3)) = e([6](13, 15), [5](7v^2, 16v^3)) \\
 &= e((13, 15), (7v^2, 16v^3))^{6 \cdot 5} \\
 &= e((13, 15), (7v^2, 16v^3))^{30} \\
 e([I]g_1, [\gamma]g_2) &= e((35, 28), (37v^2, 27v^3)) = e([4](13, 15), [4](7v^2, 16v^3)) \\
 &= e((13, 15), (7v^2, 16v^3))^{4 \cdot 4} \\
 &= e((13, 15), (7v^2, 16v^3))^{16} \\
 e([C]g_1, [\delta]g_2) &= e((27, 9), (42v^2, 16v^3)) = e([7](13, 15), [3](7v^2, 16v^3)) \\
 &= e((13, 15), (7v^2, 16v^3))^{7 \cdot 3} \\
 &= e((13, 15), (7v^2, 16v^3))^{21}
 \end{aligned}$$

In order to check equation XXX, observe that the target group \mathbb{G}_T of the Weil pairing is a finite cyclic group of order 13. Exponentiation is therefore done in modular 13 arithmetics. Using this we evaluate the left side of equation XXX as

$$e([A]g_1, [B]g_2) = e((13, 15), (7v^2, 16v^3))^{108} = e((13, 15), (7v^2, 16v^3))^4$$

since $108 \bmod 13 = 4$. Similarly, we evaluate the right side of equation XXX using modular 13 arithmetics and the exponential law $a^x \cdot a^y = a^{x+y}$. We get

$$\begin{aligned}
 &e([\alpha]g_1, [\beta]g_2) \cdot e([I]g_1, [\gamma]g_2) \cdot e([C]g_1, [\delta]g_2) = \\
 &e((13, 15), (7v^2, 16v^3))^{30} \cdot e((13, 15), (7v^2, 16v^3))^{16} \cdot e((13, 15), (7v^2, 16v^3))^{21} = \\
 &e((13, 15), (7v^2, 16v^3))^4 \cdot e((13, 15), (7v^2, 16v^3))^3 \cdot e((13, 15), (7v^2, 16v^3))^8 = \\
 &e((13, 15), (7v^2, 16v^3))^{4+3+8} = \\
 &e((13, 15), (7v^2, 16v^3))^2
 \end{aligned}$$

As we can see both the left and the right side of equation XXX are identical, which implies that the verification process accepts the simulated proof.

NOTE: UNFORTUNATELY NOT! :-((HENCE THERE IS AN ERROR SOMEWHERE ... NEED TO FIX IT AFTER VACATION)

Proof Simulation During the execution of a setup phase, a common reference string is generated accompanied by a simulation trapdoor, the latter of which must be deleted at the end of the setup-phase. As an alternative a more complicated multi-party protocol like [XXX] can be used to split the knowledge of the simulation trapdoor among many different parties.

In this paragraph we will show, why knowledge of the simulation trapdoor is problematic and how it can be used to generate zk-SNARKs for given instances without any knowledge or the existence of associated witnesses.

To be more precise, let I be an instance for some R1CS language L_R . We call a zk-SNARK for L_R **forged** or **simulated**, if it passes any verification but its generation does not require the existence of a witness W such that, $(I; W)$ is a word in L_R .

To see how simulated zk-SNARKs can be computed, assume that a forger has knowledge of proper Groth_16 parameters, a quadratic arithmetic program of the problem, a common reference string and its associated simulation trapdoor

$$\tau = (\alpha, \beta, \gamma, \delta, s) \quad (8.6)$$

Given some instance I the forgers task is to generate a zk-SNARK for this instance that passes the verification process, without access to any other zk-SNARK for this instance and without knowledge of a valid witness W .

To achieve this in the Groth_16 protocol, the forger can use the simulation trapdoor in combination with the QAP and two arbitrary field elements A and B from the scalar field \mathbb{F}_r of the pairing groups to compute

$$g_1^C = g_1^{\frac{A \cdot B}{\delta}} \cdot g_1^{-\frac{\alpha \cdot \beta}{\delta}} \cdot g_1^{-\frac{\beta A_0(s) + \alpha B_0(s) + C_0(s)}{\delta}} \cdot \left(g_1^{-\frac{\beta A_1(s) + \alpha B_1(s) + C_1(s)}{\delta}} \right)^{I_1} \cdots \left(g_1^{-\frac{\beta A_n(s) + \alpha B_n(s) + C_n(s)}{\delta}} \right)^{I_n}$$

for the instance (I_1, \dots, I_n) . The forger then publishes the zk-SNARK $\pi_{forged} = (g_1^A, g_1^C, g_2^B)$, which will pass the verification process and is computable without the existence of a witness (W_1, \dots, W_m) .

To see that the simulation trapdoor is necessary and sufficient to compute the simulated proof π_{forged} , first observe that both generators g_1 and g_2 are known to the forger, as they are part of the common reference string, encoded as $g_1^{s^0}$ and $g_2^{s^0}$. The forger is therefore able to compute $g_1^{A \cdot B}$. Moreover, since the forger knows α, β, δ and s from the trapdoor, they are able to compute all factors in the computation of g_1^C .

If, on the other hand, the simulation trapdoor is unknown, it is not possible to compute g_1^C , since for example the computational Diffie-Hellman assumption makes the derivation of $g_1^{\alpha \cdot \beta}$ from g_1^α and g_1^β infeasible.

Example 140 (The 3-factorization Problem). To see how a forger might simulate a zk-SNARK for some given instance I , consider the 3-factorization problem from XXX, our protocol parameters from XXX, the common reference string from XXX and the simulation trapdoor $\tau = (6, 5, 4, 3, 2)$ of that CRS.

In order to forge a zk-SNARK for instance $I_1 = 11$ we don't need a constructive proof for the associated rank-1 constraints system, which implies that we don't have to execute the circuit $C_{3, fac}(\mathbb{F}_{13})$. Instead we have to choose 2 arbitrary elements A and B from \mathbb{F}_{13} and compute g_1^A , g_2^B and g_1^C as defined in XXX. We choose $A = 9$ and $B = 3$ and since $\delta^{-1} = 3$, we compute

$$\begin{aligned} [A]g_1 &= [9](13, 15) = (35, 15) \\ [B]g_2 &= [3](7v^2, 16v^3) = (42v^2, 16v^3) \\ [C]g_1 &= \left[\frac{A \cdot B}{\delta} \right] g_1 \oplus \left[-\frac{\alpha \cdot \beta}{\delta} \right] g_1 \oplus \left[-\frac{\beta A_0(s) + \alpha B_0(s) + C_0(s)}{\delta} \right] g_1 \oplus \\ &\quad [I_1] \left[-\frac{\beta A_1(s) + \alpha B_1(s) + C_1(s)}{\delta} \right] g_1 \\ &= [(9 \cdot 3) \cdot 9](13, 15) \oplus [-(6 \cdot 5) \cdot 9](13, 15) \oplus [0](13, 15) \oplus [11][-(7 \cdot 2 + 4) \cdot 9](13, 15) \\ &= [9](13, 15) \oplus [3](13, 15) \oplus [12](13, 15) = [11](13, 15) \\ &= (33, 9) \end{aligned}$$

This is all we need to generate our forged proof for the 3-factorization problem. We publish the simulated zk-SNARK

$$\pi_{fake} = ((35, 15), (33, 9), (42v^2, 16v^3))$$

add reference

add reference

add reference

add reference

5421 Despite the fact that this zk-SNARK was generated without knowledge of a proper witness, it
 5422 is indistinguishable from a zk-SNARK that proofs knowledge of a proper witness.

To see that we show that our forged SNARK passes the verification process. In order to verify π_{fake} we proceed as in XXX and compute the curve point g_1^I for the instance $I_1 = 11$. Since the instance is the same as in example XXX, we can parallel the computation from XXX and get

$$\begin{aligned} [I]g_1 &= \left[\frac{\beta \cdot A_0(s) + \alpha \cdot B_0(s) + C_0(s)}{\gamma} \right]_{g_1} \oplus [I_1] \left[\frac{\beta \cdot A_1(s) + \alpha \cdot B_1(s) + C_1(s)}{\gamma} \right]_{g_1} \\ &= (35, 28) \end{aligned}$$

In a next step we have to compute all the pairings involved in equation XXX. Using the logarithmic order on \mathbb{G}_1 and \mathbb{G}_2 as well as the bilinearity property of the pairing map we get

$$\begin{aligned} e([A]g_1, [B]g_2) &= e((35, 15), (42v^2, 16v^3)) = e([9](13, 15), [3](7v^2, 16v^3)) \\ &= e((13, 15), (7v^2, 16v^3))^{9 \cdot 3} \\ &= e((13, 15), (7v^2, 16v^3))^{27} \\ e([\alpha]g_1, [\beta]g_2) &= e((27, 34), (16v^2, 28v^3)) = e([6](13, 15), [5](7v^2, 16v^3)) \\ &= e((13, 15), (7v^2, 16v^3))^{6 \cdot 5} \\ &= e((13, 15), (7v^2, 16v^3))^{30} \\ e([I]g_1, [\gamma]g_2) &= e((35, 28), (37v^2, 27v^3)) = e([4](13, 15), [4](7v^2, 16v^3)) \\ &= e((13, 15), (7v^2, 16v^3))^{4 \cdot 4} \\ &= e((13, 15), (7v^2, 16v^3))^{16} \\ e([C]g_1, [\delta]g_2) &= e((33, 9), (42v^2, 16v^3)) = e([11](13, 15), [3](7v^2, 16v^3)) \\ &= e((13, 15), (7v^2, 16v^3))^{11 \cdot 3} \\ &= e((13, 15), (7v^2, 16v^3))^{33} \end{aligned}$$

In order to check equation XXX, observe that the target group \mathbb{G}_T of the Weil pairing is a finite cyclic group of order 13. Exponentiation is therefore done in modular 13 arithmetics. Using this we evaluate the left side of equation XXX as

$$e([A]g_1, [B]g_2) = e((13, 15), (7v^2, 16v^3))^{27} = e((13, 15), (7v^2, 16v^3))^1$$

since $27 \bmod 13 = 1$. Similarly, we evaluate the right side of equation XXX using modular 13 arithmetics and the exponential law $a^x \cdot a^y = a^{x+y}$. We get

$$\begin{aligned} e([\alpha]g_1, [\beta]g_2) \cdot e([I]g_1, [\gamma]g_2) \cdot e([C]g_1, [\delta]g_2) &= \\ e((13, 15), (7v^2, 16v^3))^{30} \cdot e((13, 15), (7v^2, 16v^3))^{16} \cdot e((13, 15), (7v^2, 16v^3))^{33} &= \\ e((13, 15), (7v^2, 16v^3))^4 \cdot e((13, 15), (7v^2, 16v^3))^3 \cdot e((13, 15), (7v^2, 16v^3))^7 &= \\ e((13, 15), (7v^2, 16v^3))^{4+3+7} &= \\ e((13, 15), (7v^2, 16v^3))^1 & \end{aligned}$$

5423 As we can see both the left and the right side of equation XXX are identical, which implies that
 5424 the verification process accepts the simulated proof. π_{fake} therefore convince the verifier that
 5425 a witness to 3-factorization problem exists, However, no such witness was really necessary to
 5426 generate the proof.

5427 **Chapter 9**

5428 **Exercises and Solutions**

5429 TODO: All exercises we provided should have a solution, which we give here in all detail.

Bibliography

- Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016. URL <http://eprint.iacr.org/2016/260>.
- David Fifield. The equivalence of the computational diffie–hellman and discrete logarithm problems in certain groups, 2012. URL <https://web.stanford.edu/class/cs259c/finalpapers/dlp-cdh.pdf>.
- Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. URL <https://fmouhart.epHEME.re/Crypto-1617/TD08.pdf>.
- Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. Cryptology ePrint Archive, Report 2016/492, 2016. <https://ia.cr/2016/492>.