
Operational notes

Document updated on **March 26, 2022**.

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

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

11 Todo list

12	zero-knowledge proofs	12
13	played with	12
14	finite field	12
15	elliptic curve	12
16	Update reference when content is finalized	12
17	methatical	12
18	add some more informal explanation of absolute value	14
19	We haven't really talked about what a ring is at this point	14
20	What's the significance of this distinction?	15
21	reverse	15
22	Turing machine	15
23	polynomial time	15
24	sub-exponentially, with $\mathcal{O}((1 + \varepsilon)^n)$ and some $\varepsilon > 0$	15
25	Add text	16
26	\mathbb{Q} of fractions	16
27	Division in the usual sense is not defined for integers	16
28	Add more explanation of how this works	17
29	pseudocode	18
30	check reference	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	add reference	28
46	rephrase	32
47	subtrahend	33
48	minuend	33
49	what does this mean?	37
50	Def Subgroup, Fundamental theorem of cyclic groups.	40

51	add reference	41
52	Add real-life example of 0?	41
53	add reference	41
54	check reference	42
55	check references to previous examples	43
56	RSA crypto system	43
57	size 2048-bits	43
58	check reference	43
59	add reference: 27?	43
60	check reference	44
61	polynomial time	44
62	exponential time	44
63	TODO: Fundamental theorem of finite cyclic groups	44
64	check reference	44
65	runtime complexity	45
66	add reference	45
67	S: what does “efficiently” mean here?	45
68	computational hardness assumptions	45
69	check reference	45
70	check reference	46
71	explain last sentence more	46
72	“equation”?	47
73	check reference	47
74	what’s the difference between \mathbb{F}_p^* and \mathbb{Z}_p^* ?	47
75	Legendre symbol	47
76	Euler’s formular	47
77	These are only explained later in the text, “	47
78	are these going to be relevant later?	48
79	TODO: theorem: every factor of order defines a subgroup...	48
80	Is there a term for this property?	49
81	a few examples?	51
82	check reference	51
83	TODO: DOUBLE CHECK THIS REASONING.	51
84	Mirco: We can do better than this	53
85	check reference	54
86	add reference	55
87	pseudorandom	55
88	oracle	55
89	check reference	55
90	add text on this	55
91	check reference	57
92	check reference	57
93	check reference	57
94	check reference	58
95	add more examples protocols of SNARK	58
96	check reference	58
97	add reference	58
98	Abelian groups	58

99	■ codomain	58
100	■ Check change of wording	59
101	■ add reference	60
102	■ Expand on this?	60
103	■ check reference	60
104	■ unify \mathbb{Z}_5 and \mathbb{F}_5 across the board?	61
105	■ S: are we introducing elliptic curves in section 1 or 2?	61
106	■ check reference	62
107	■ check reference	62
108	■ add reference	62
109	■ check reference	62
110	■ write paragraph on exponentiation	63
111	■ add reference	63
112	■ check reference	63
113	■ add reference	63
114	■ group pairings	63
115	■ add reference	64
116	■ check reference	64
117	■ check reference	67
118	■ add reference	68
119	■ TODO: Elliptic Curve asymmetric cryptography examples. Private key, generator,	
120	public key.	70
121	■ add reference	70
122	■ maybe remove this sentence?	70
123	■ affine space	70
124	■ cusps	71
125	■ self-intersections	71
126	■ check reference	72
127	■ check reference	73
128	■ jubjub	73
129	■ check reference	73
130	■ affine plane	73
131	■ check reference	74
132	■ check reference	74
133	■ check reference	75
134	■ sign	75
135	■ more explanation of what the sign is	75
136	■ check reference	75
137	■ S: I don't follow this at all	76
138	■ check reference	76
139	■ add explanation of how this shows what we claim	76
140	■ should this def. be moved even earlier?	77
141	■ chord line	77
142	■ tangential	77
143	■ tangent line	77
144	■ remove Q ?	77
145	■ where?	78
146	■ check reference	78

















































147	check reference	78
148	check reference	78
149	check reference	79
150	check reference	79
151	check reference	80
152	check reference	80
153	check reference	81
154	add term	81
155	add term	81
156	add reference	81
157	cofactor clearing	81
158	add reference	81
159	check reference	81
160	check reference	82
161	add reference	82
162	add reference	82
163	check reference	82
164	check reference	82
165	check reference	83
166	check reference	83
167	check reference	83
168	Explain how	83
169	write example	84
170	check reference	84
171	add reference	84
172	check reference	84
173	add reference	85
174	check reference	85
175	add reference	85
176	check reference	85
177	add reference	85
178	check reference	85
179	add reference	85
180	add reference	85
181	add reference	85
182	check reference	85
183	check reference	85
184	Check if following Alg is floated too far	86
185	add reference	86
186	add reference	86
187	write up this part	86
188	is the label in L ^A T _E X correct here?	88
189	check reference	88
190	check reference	88
191	check reference	88
192	check reference	89
193	check reference	89
194	check reference	90

















































195	■ check reference	90
196	■ check reference	90
197	■ check reference	90
198	■ check reference	90
199	■ add reference	90
200	■ check reference	92
201	■ check reference	92
202	■ check reference	92
203	■ check reference	92
204	■ check reference	92
205	■ change “tiny-jubjub” to “pen-jubjub” throughout?	93
206	■ check reference	94
207	■ check reference	94
208	■ check reference	95
209	■ either expand on this or delete it	95
210	■ add reference	95
211	■ check reference	95
212	■ check reference	95
213	■ check reference	95
214	■ check reference	95
215	■ check reference	95
216	■ check reference	96
217	■ check reference	96
218	■ check reference	97
219	■ check reference	97
220	■ add reference	97
221	■ add reference	97
222	■ This needs to be written (in Algebra)	97
223	■ add reference	97
224	■ add reference	97
225	■ check reference	97
226	■ towers of curve extensions	98
227	■ check reference	98
228	■ check reference	98
229	■ check reference	98
230	■ check reference	99
231	■ add reference	99
232	■ is “huge” a technical term?	99
233	■ check reference	100
234	■ S: either add more explanation or move to a footnote	100
235	■ type 3 pairing-based cryptography	100
236	■ add references?	100
237	■ check reference	101
238	■ check reference	101
239	■ check floating of algorithm	102
240	■ add references	102
241	■ check reference	103
242	■ add reference	103





















243	check reference	103
244	check reference	103
245	add reference	104
246	should all lines of all algorithms be numbered?	104
247	check reference	105
248	check reference	105
249	check reference	105
250	check if the algorithm is floated properly	105
251	check reference	105
252	again?	107
253	check reference	107
254	circuit	107
255	signature schemes	107
256	this was called “pen-jubjub”.	107
257	check reference	107
258	add reference	108
259	check reference	108
260	add references	108
261	add reference	108
262	reference text to be written in Algebra	108
263	check reference	108
264	check reference	108
265	check reference	109
266	add reference	109
267	algebraic closures	109
268	check reference	109
269	check reference	110
270	check reference	110
271	check reference	110
272	check reference	111
273	disambiguate	111
274	add reference	111
275	unify terminology	111
276	check reference	112
277	actually make this a table?	112
278	exercise still to be written?	113
279	add reference	113
280	check reference	113
281	check reference	113
282	add reference	114
283	check reference	115
284	check reference	115
285	check reference	115
286	add reference	116
287	check reference	116
288	check reference	116
289	check reference	117
290	what does this mean?	118

291	write up this part	119
292	add reference	119
293	check reference	119
294	cyclotomic polynomial	119
295	Pholaard-rho attack	119
296	todo	119
297	why?	120
298	check reference	120
299	check reference	120
300	what does this mean?	120
301	add reference	120
302	add reference	120
303	check reference	120
304	check reference	121
305	add reference	122
306	add exercise	122
307	check reference	123
308	add reference	123
309	add reference	123
310	add reference	123
311	check reference	124
312	check reference	124
313	add reference	124
314	add reference	124
315	add reference	125
316	check reference	125
317	add reference	125
318	add reference	125
319	finish writing this up	126
320	add reference	126
321	correct computations	126
322	fill in missing parts	126
323	add reference	127
324	check equation	127
325	Chapter 1?	128
326	"rigorous"?	128
327	"proving"?	128
328	Add example	129
329	Add more explanation	129
330	I'd delete this, too distracting	129
331	binary tuples	129
332	add reference	130
333	add reference	130
334	check reference	130
335	check reference	130
336	Are we using w and x interchangeably or is there a difference between them?	131
337	check reference	131
338	jubjub	131

339	check reference	131
340	check reference	131
341	check wording	131
342	check reference	131
343	check references	132
344	add reference	132
345	add reference	132
346	check reference	133
347	add reference	133
348	check reference	134
349	check reference	134
350	add reference	135
351	add reference	136
352	Schur/Hadamard product	136
353	add reference	136
354	check reference	136
355	check reference	137
356	add reference	138
357	check reference	139
358	check reference	139
359	check reference	139
360	check reference	139
361	check reference	140
362	add reference	140
363	add reference	141
364	check reference	141
365	check reference	142
366	check reference	142
367	check reference	142
368	add reference	143
369	check reference	145
370	add reference	145
371	check reference	146
372	check reference	146
373	check reference	146
374	Should we refer to R1CS satisfiability (p. 139 here?	147
375	check reference	148
376	add reference	148
377	add reference	148
378	check reference	149
379	check reference	149
380	check reference	150
381	check reference	152
382	add reference	153
383	"by"?	153
384	check reference	153
385	check reference	153
386	add reference	153

387	 add reference	153
388	 check reference	153
389	 add reference	153
390	 clarify language	155
391	 check reference	156
392	 add reference	156
393	 check reference	156
394	 add reference	156
395	 add references	159
396	 add references to these languages?	159
397	 add reference	162
398	 add reference	163
399	 add reference	163
400	 add reference	164
401	 add reference	165
402	 add reference	165
403	 add reference	167
404	 add reference	167
405	 add reference	168
406	 add reference	168
407	 add reference	168
408	 add reference	168
409	 add reference	168
410	 add reference	169
411	 add reference	169
412	 add reference	169
413	 add reference	169
414	 add reference	169
415	 add reference	170
416	 add reference	171
417	 "constraints" or "constrained"?	171
418	 add reference	172
419	 "constraints" or "constrained"?	172
420	 add reference	172
421	 "constraints" or "constrained"?	172
422	 add reference	173
423	 add reference	173
424	 add reference	173
425	 add reference	173
426	 add reference	174
427	 add reference	175
428	 add reference	175
429	 add reference	175
430	 shift	177
431	 bishift	178
432	 add reference	179
433	 add reference	180
434	 something missing here?	181

435		add reference	182
436		add reference	183
437		add reference	184
438		add reference	184
439		add reference	184
440		add reference	185
441		add reference	185
442		add reference	185
443		add reference	186
444		add reference	187
445		add reference	188
446		add reference	188
447		add reference	188
448		add reference	189
449		add reference	189
450		add reference	189
451		add reference	189
452		add reference	189
453		"invariable"?	189
454		add reference	190
455		add reference	190
456		add reference	190
457		add reference	191
458		add reference	191
459		add reference	192
460		add reference	193
461		add reference	193
462		add reference	194
463		add reference	194
464		add reference	194
465		add reference	194
466		add reference	194
467		add reference	195
468		add reference	195
469		add reference	195
470		add reference	195
471		add reference	195
472		add reference	195
473		add reference	195
474		add reference	195
475		add reference	195
476		add reference	196
477		add reference	196
478		add reference	196
479		add reference	196
480		add reference	198
481		add reference	198
482		add reference	198

483		add reference	198
484		add reference	198
485		add reference	198
486		add reference	199
487		add reference	199
488		add reference	199
489		add reference	199
490		add reference	199
491		add reference	200
492		add reference	200
493		add reference	200
494		add reference	200
495		add reference	201
496		add reference	201
497		add reference	201
498		add reference	201
499		add reference	201
500		add reference	201
501		add reference	201
502		add reference	201

503

MoonMath manual

504

TechnoBob and the Least Scruples crew

505

March 26, 2022

Contents

507	1 Introduction	5
508	1.1 Target audience	5
509	1.2 The Zoo of Zero-Knowledge Proofs	6
510	To Do List	8
511	Points to cover while writing	8
512	2 Preliminaries	9
513	2.1 Preface and Acknowledgements	9
514	2.2 Purpose of the book	9
515	2.3 How to read this book	10
516	2.4 Cryptological Systems	10
517	2.5 SNARKS	10
518	2.6 complexity theory	10
519	2.6.1 Runtime complexity	10
520	2.7 Software Used in This Book	11
521	2.7.1 Sagemath	11
522	3 Arithmetics	12
523	3.1 Introduction	12
524	3.1.1 Aims and target audience	12
525	3.1.2 The structure of this chapter	13
526	3.2 Integer Arithmetics	13
527	Euclidean Division	16
528	The Extended Euclidean Algorithm	18
529	3.3 Modular arithmetic	19
530	Conguency	20
531	Modular Arithmetics	20
532	The Chinese Remainder Theorem	23
533	Modular Inverses	26
534	3.4 Polynomial Arithmetics	29
535	Polynomial Arithmetics	33
536	Euklidean Division	34
537	Prime Factors	36
538	Lange interpolation	37
539	4 Algebra	40
540	4.1 Groups	40
541	Commutative Groups	41
542	Finite groups	43

543		Generators	43
544		The discrete Logarithm problem	43
545	4.1.1	Cryptographic Groups	44
546		The discrete logarithm assumption	45
547		The decisional Diffie–Hellman assumption	47
548		The computational Diffie–Hellman assumption	47
549		Cofactor Clearing	48
550	4.1.2	Hashing to Groups	48
551		Hash functions	48
552		Hashing to cyclic groups	50
553		Hashing to modular arithmetics	51
554		Pedersen Hashes	55
555		MimC Hashes	55
556		Pseudorandom Functions in DDH-A groups	55
557	4.2	Commutative Rings	55
558		Hashing to Commutative Rings	58
559	4.3	Fields	58
560		Prime fields	60
561		Square Roots	61
562		Exponentiation	63
563		Hashing into prime fields	63
564		Extension Fields	63
565		Hashing into extension fields	67
566	4.4	Projective Planes	67
567	5	Elliptic Curves	70
568	5.1	Elliptic Curve Arithmetics	70
569	5.1.1	Short Weierstraß Curves	70
570		Affine short Weierstraß form	71
571		Affine compressed representation	75
572		Affine group law	76
573		Scalar multiplication	81
574		Projective short Weierstraß form	84
575		Projective Group law	85
576		Coordinate Transformations	86
577	5.1.2	Montgomery Curves	86
578		Affine Montgomery Form	88
579		Affine Montgomery coordinate transformation	89
580		Montgomery group law	91
581	5.1.3	Twisted Edwards Curves	92
582		Twisted Edwards Form	92
583		Twisted Edwards group law	94
584	5.2	Elliptic Curve Pairings	95
585		Embedding Degrees	95
586		Elliptic Curves over extension fields	96
587		Full torsion groups	97
588		Torsion subgroups	100
589		The Weil pairing	102

590	5.3	Hashing to Curves	104
591		Try-and-increment hash functions	105
592	5.4	Constructing elliptic curves	107
593		The Trace of Frobenius	108
594		The j -invariant	109
595		The Complex Multiplication Method	110
596		The <i>BLS6_6</i> pen-and-paper curve	119
597		Hashing to pairing groups	126
598	6	Statements	128
599	6.1	Formal Languages	128
600		Decision Functions	129
601		Instance and Witness	132
602		Modularity	135
603	6.2	Statement Representations	135
604	6.2.1	Rank-1 Quadratic Constraint Systems	135
605		R1CS representation	136
606		R1CS Satisfiability	138
607		Modularity	140
608	6.2.2	Algebraic Circuits	140
609		Algebraic circuit representation	140
610		Circuit Execution	145
611		Circuit Satisfiability	147
612		Associated Constraint Systems	148
613	6.2.3	Quadratic Arithmetic Programs	153
614		QAP representation	153
615		QAP Satisfiability	155
616	7	Circuit Compilers	159
617	7.1	A Pen-and-Paper Language	159
618	7.1.1	The Grammar	159
619	7.1.2	The Execution Phases	161
620		The Setup Phase	161
621		The Prover Phase	163
622	7.2	Common Programing concepts	163
623	7.2.1	Primitive Types	163
624		The base-field type	164
625		The Subtraction Constraint System	167
626		The Inversion Constraint System	168
627		The Division Constraint System	169
628		The boolean Type	170
629		The boolean Constraint System	170
630		The AND operator constraint system	171
631		The OR operator constraint system	171
632		The NOT operator constraint system	172
633		Modularity	173
634		Arrays	176
635		The Unsigned Integer Type	176

636		The uN Constraint System	177
637		The Unsigned Integer Operators	178
638	7.2.2	Control Flow	179
639		The Conditional Assignment	179
640		Loops	181
641	7.2.3	Binary Field Representations	182
642	7.2.4	Cryptographic Primitives	184
643		Twisted Edwards curves	184
644		Twisted Edwards curves constraints	184
645		Twisted Edwards curve addition	185
646	8	Zero Knowledge Protocols	186
647	8.1	Proof Systems	186
648	8.2	The “Groth16” Protocol	187
649		The Setup Phase	189
650		The Proofer Phase	194
651		The Verification Phase	197
652		Proof Simulation	199
653	9	Exercises and Solutions	202

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.

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.

One might argue that the understanding of formal languages is not very important in SNARK development and associated statement design, but terms from that field of research are standard jargon in many papers on zero-knowledge proofs. We therefore believe that at least some introduction to formal languages and how they fit into the picture of SNARK development is beneficial, mostly to give developers a better intuition about where all this is located in the bigger picture of the logic landscape. In addition, formal languages give a better understanding of what a formal proof for a statement actually is.

Roughly speaking, a formal language (or just language for short) is nothing but a set of words, *th*. Words, in turn, are strings of letters taken from some alphabet and formed according to some defining rules of the language.

To be more precise, let Σ be any set and Σ^* the set of all finite **tuples** (ordered lists) (x_1, \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,

Chapter
1?

"rigorous"?

"proving"?

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 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 are given below:

binary tuples

- 4147 • $R(1, 0, 1) = \text{true}$,
- 4148 • $R(0) = \text{true}$,
- 4149 • $R() = \text{true}$,
- 4150 • but $R(1, 1) = \text{false}$.

4151 Inside our language L_{alt} , it makes sense to claim the following statement: “There exists an
 4152 alternating string.” One way to prove this statement constructively is by providing an actual
 4153 instance, that is, finding actual alternating string like $x = (1, 0, 1)$. Constructing string $(1, 0, 1)$
 4154 therefore proves the statement “There exists an alternating string.”, because it is easy to verify
 4155 that $R(1, 0, 1) = \text{true}$.

4156 *Example 104 (Programming Language).* Programming languages are a very important class of
 4157 formal languages. For these languages, the alphabet is usually (a subset) of the ASCII table,
 4158 and the grammar is defined by the rules of the programming language’s compiler. Words, then,
 4159 are nothing but properly written computer programs that the compiler accepts. The compiler
 4160 can therefore be interpreted as the decision function.

4161 To give an unusual example strange enough to highlight the point, consider the program-
 4162 ming language Malbolge as defined in XXX. This language was specifically designed to be
 4163 almost impossible to use and writing programs in this language is a difficult task. An inter-
 4164 esting claim is therefore the statement: “There exists a computer program in Malbolge”. As it
 4165 turned out, proving this statement constructively, that is, by providing an actual instance of such
 4166 a program, was not an easy task, as it took two years after the introduction of Malbolge to write
 4167 a program that its compiler accepts. So, for two years, no one was able to prove the statement
 4168 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^WV>gY;dts76qKJImZkj

4169 is an example of such a proof, as it is excepted by the Malbolge compiler and is compiled to
 4170 an executable binary that displays “Hello, World.” (See XXX). In this example, the Malbolge
 4171 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

$$R_\emptyset : \mathbb{Z}_6^* \rightarrow \{\text{true}, \text{false}\} ; (x_1, \dots, x_n) \mapsto \begin{cases} \text{true} & n = 4 \text{ and } x_1 \cdot x_1 = 2 \\ \text{true} & \text{else} \end{cases}$$

4172 We write L_\emptyset for the associated language. As we can see from the multiplication table of \mathbb{Z}_6
 4173 in example 8 in chapter 3, the ring \mathbb{Z}_6 does not contain any element x such that $x^2 = 2$, which
 4174 implies $R_\emptyset(x_1, \dots, x_n) = \text{false}$ for all tuples $(x_1, \dots, x_n) \in \Sigma^*$. The language therefore does
 4175 not contain any words. Proving the statement “There exists a word in L_\emptyset ” constructively by
 4176 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 (see section 5.1.3).

In the first step, we define a language such that points on the tiny jubjub curve are in 1:1 correspondence with words in that language.

Since the tiny jubjub curve is an elliptic curve over the field \mathbb{F}_{13} , we choose the alphabet $\Sigma = \mathbb{F}_{13}$. In this case, the set \mathbb{F}_{13}^* consists of all finite strings of field elements from \mathbb{F}_{13} . To define the grammar, recall from 66 that a point on the tiny jubjub curve is a pair (x, y) of field elements such that $3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$. We can use this equation to derive the following decision function:

$$R_{tiny.jj} : \mathbb{F}_{13}^* \rightarrow \{true, false\} ; (x_1, \dots, x_n) \mapsto \begin{cases} true & n = 2 \text{ and } 3 \cdot x_1^2 + x_2^2 = 1 + 8 \cdot x_1^2 \cdot x_2^2 \\ false & else \end{cases}$$

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 66 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

check reference

check reference

check wording

check reference

4207 and the computation $R_{tiny.jj}(11,6) = true$ is a proof verification. In contrast, the tuple $(1,1)$ is
 4208 not a proof of the statement, since the check $R_{tiny.jj}(1,1) = false$ does not verify the proof.

4209 *Exercise 39.* Consider exercise XXX again. Define a decision function such that the associated
 4210 language $L_{Exercise_{XXX}}$ consist precisely of all solutions to the equation $5x + 4 = 28 + 2x$ over
 4211 \mathbb{F}_{13} . Provide a constructive proof for the claim: “There exist a word in $L_{Exercise_{XXX}}$ and verify
 4212 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

$$R_{example_8} : \Sigma^* \rightarrow \{true, false\} ; x \mapsto \begin{cases} true & x.len() = 1 \text{ and } 3 \cdot x + 3 = 0 \\ false & \text{else} \end{cases}$$

4213 Compute all words in the associated language $L_{example_8}$, provide a constructive proof for the
 4214 statement “There exist a word in $L_{example_example_8}$ ” and verify the proof.
 4215

check
references

4216 **Instance and Witness** As we have seen in the previous paragraph, statements provide mem-
 4217 bership claims in formal languages, and instances serve as constructive proofs for those claims.
 4218 However, in the context of **zero-knowledge** proof systems, our naive notion of constructive
 4219 proofs is refined in such a way that its possible to hide parts of the proof instance and still be
 4220 able to prove the statement. In this context, it is therefore necessary to split a proof into a **public**
 4221 **part** called the **instance** and a private part called a **witness**.

4222 To account for this separation of a proof instance into a public and a private part, our previ-
 4223 ous definition of formal languages needs a refinement in the context of zero-knowledge proof
 4224 systems. Instead of a single alphabet, the refined definition considers two alphabets Σ_I and Σ_W ,
 4225 and a decision function defined as follows:

$$R : \Sigma_I^* \times \Sigma_W^* \rightarrow \{true, false\} ; (i; w) \mapsto R(i; w) \quad (6.3)$$

4226 Words are therefore tuples $(i; w) \in \Sigma_I^* \times \Sigma_W^*$ with $R(i; w) = true$. The refined definition differ-
 4227 entiates between public inputs $i \in \Sigma_I$ and private inputs $w \in \Sigma_W$. The public input i is called an
 4228 **instance** and the private input w is called a **witness** of R .

4229 If a decision function is given, the associated language is defined as the set of all tuples from
 4230 the underlying alphabet that are verified by the decision function:

$$L_R := \{(i; w) \in \Sigma_I^* \times \Sigma_W^* \mid R(i; w) = true\} \quad (6.4)$$

4231 In this refined context, a **statement** S is a claim that, given an instance $i \in \Sigma_I^*$, there is a witness
 4232 $w \in \Sigma_W^*$ such that language L contains a word $(i; w)$. A constructive **proof** for statement S is
 4233 given by some string $P = (i; w) \in \Sigma_I^* \times \Sigma_W^*$ and a proof is **verified** by checking $R(P) = true$.

4234 It is worth understanding the difference between statements as defined in XXX and the
 4235 refined notion of statements from this paragraph. While statements in the sense of the previous
 4236 paragraph can be seen as membership claims, statements in the refined definition can be seen
 4237 as knowledge-proofs, where a prover claims knowledge of a witness for a given instance. For a
 4238 more detailed discussion on this topic see [XXX sec 1.4]

add refer-
ence

add refer-
ence

4239 *Example 108* (SHA256 – Knowledge of Preimage). One of the most common examples in the
 4240 context of zero-knowledge proof systems is the **knowledge-of-a-preimage proof** for some
 4241 cryptographic hash function like *SHA256*, where a publicly known *SHA256* digest value is

given, and the task is to prove knowledge of a preimage for that digest under the $SHA256$ function, without revealing that preimage.

To understand this problem in detail, we have to introduce a language able to describe the knowledge-of-preimage problem in such a way that the claim “Given digest i , there is a preimage w such that $SHA256(w) = i$ ” becomes a statement in that language. Since $SHA256$ is a function

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

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.

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.

We can formalize this new language, which we might call L_{3, fac_zk} , by defining the following decision function:

$$R_{3, fac_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\};$$

$$((i_1, \dots, i_n); (w_1, \dots, w_m)) \mapsto \begin{cases} true & n = 1, m = 3, i_1 = w_1 \cdot w_2 \cdot w_3 \\ false & else \end{cases}$$

check
reference

add refer-
ence

4270 The associated language $L_{3.fac_zk}$ is defined by all tuples from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto
 4271 *true* under the decision function $R_{3.fac_zk}$.

4272 Considering the distinction we made between the instance and the witness part in $L_{3.fac_zk}$,
 4273 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
 4274 be the instance and why we didn't choose another combination? This was an arbitrary choice
 4275 in the example. Every other combination of private and public factors would be equally valid.
 4276 For example, it would be possible to declare all variables as private or to declare all variables as
 4277 public. Actual choices are determined by the application only.

4278 *Example 110 (The Tiny JubJub Curve).* Consider the language $L_{tiny.jj}$ from example 107. As
 4279 we have seen, a constructive proof in $L_{tiny.jj}$ is given by a pair (x_1, x_2) of field elements from
 4280 \mathbb{F}_{13} such that the pair is a point of the tiny jubjub curve in its Edwards representation.

check
reference

4281 We look at a reasonable splitting of words from $L_{tiny.jj}$ into private and public parts. The
 4282 two obvious choices are to either choose both coordinates x_1 as x_2 as public inputs, or to choose
 4283 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:

$$R_{tiny.jj.1} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\};$$

$$(I_1, \dots, I_n; W_1, \dots, W_m) \mapsto \begin{cases} true & n = 2, m = 0 \text{ and } 3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2 \\ false & \text{else} \end{cases}$$

4284 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
 4285 *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:

$$R_{tiny.jj_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\};$$

$$(I_1, \dots, I_n; W_1, \dots, W_m) \mapsto \begin{cases} true & n = 0, m = m \text{ and } 3 \cdot W_1^2 + W_2^2 = 1 + 8 \cdot W_1^2 \cdot W_2^2 \\ false & \text{else} \end{cases}$$

4286 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
 4287 *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

$$R_{linear} : \Sigma^* \times \Sigma^* \rightarrow \{true, false\};$$

$$(i; w) \mapsto \begin{cases} true & i.len() = 3 \text{ and } w.len() = 1 \text{ and } i_1 \cdot w_1 + i_2 = i_3 \\ false & \text{else} \end{cases}$$

4288 Which of the following instances (i_1, i_2, i_3) has a proof of knowledge in L_{linear} ?

- 4289 • $(3, 3, 0)$
- 4290 • $(2, 1, 0)$
- 4291 • $(4, 4, 2)$

add reference

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 $L_1 \cap L_2$:

$$R_{L_1 \cap L_2} : \Sigma_I^* \times \Sigma_W^* \rightarrow \{true, false\}; (i, w) \mapsto R_1(i, w) \text{ and } R_2(i, w) \quad (6.6)$$

Thus, the intersection of two decision-function-based languages is a also decision-function-based language. This is important from an implementations point of view: It allows us to construct complex decision functions, their languages and associated statements from simple building blocks. Given a publicly known instance $i \in \Sigma_I^*$ a statement in an intersection language then claims knowledge of a witness that satisfies all relations simultaneously.

6.2 Statement Representations

As we have seen in the previous section, formal languages and their definitions by decision functions are a powerful tool to describe statements in a formally rigorous manner.

However, from the perspective of existing zero-knowledge proof systems, not all ways to actually represent decision functions are equally useful. Depending on the proof system, some are more suitable than others. In this section, will describe two of the most common ways to represent decision functions and their statements.

6.2.1 Rank-1 Quadratic Constraint Systems

Although decision functions are expressible in various ways, many contemporary proofing systems require the deciding relation to be expressed in terms of a system of quadratic equations

over a finite field. This is true in particular for pairing-based proofing systems like XXX, roughly because it is possible to check solutions to those equations “in the exponent” of pairing-friendly cryptographic groups.

add reference

In this section, we will therefore have a closer look at a particular type of quadratic equation called **rank-1 quadratic constraints systems**, which are a common standard in zero-knowledge proof systems. We will start with a general introduction to those systems and then look at their relation to formal languages. We will look into a common way to compute solutions to those systems, and then show how a simple compiler might derive rank-1 constraint systems from more high-level programming code.

R1CS representation To understand what **rank-1 (quadratic) constraint systems** are in detail, let \mathbb{F} be a field, n, m and $k \in \mathbb{N}$ three numbers and a_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:

Definition 6.2.1.1. R1CS representation

$$\begin{aligned} (a_0^1 + \sum_{j=1}^n a_j^1 \cdot I_j + \sum_{j=1}^m a_{n+j}^1 \cdot W_j) \cdot (b_0^1 + \sum_{j=1}^n b_j^1 \cdot I_j + \sum_{j=1}^m b_{n+j}^1 \cdot W_j) &= c_0^1 + \sum_{j=1}^n c_j^1 \cdot I_j + \sum_{j=1}^m c_{n+j}^1 \cdot W_j \\ &\vdots \\ (a_0^k + \sum_{j=1}^n a_j^k \cdot I_j + \sum_{j=1}^m a_{n+j}^k \cdot W_j) \cdot (b_0^k + \sum_{j=1}^n b_j^k \cdot I_j + \sum_{j=1}^m b_{n+j}^k \cdot W_j) &= c_0^k + \sum_{j=1}^n c_j^k \cdot I_j + \sum_{j=1}^m c_{n+j}^k \cdot W_j \end{aligned}$$

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

$$Ax \odot Bx = Cx$$

Schur/Hadamard product

However, since we did not introduced matrix calculus in the book, we use XXX as the defining equation for rank-1 constraints systems. We only highlighted the matrix notation, because it is sometimes used in the literature.

add reference

Generally speaking, the idea of a rank-1 constraint system is to keep track of all the values that any variable can assume during a computation and to bind the relationships among all those variables that are implied by the computation itself. Enforcing relations between all the steps of a computer program, the execution is then constrained to be computed in exactly the expected way without any opportunity for deviations. In this sense, solutions to rank-1 constraint systems are proofs of proper 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

$$W_1 \cdot W_2 = W_4 \quad \text{constraint 1}$$

$$W_4 \cdot W_3 = I_1 \quad \text{constraint 2}$$

4359 Given some instance I_1 , any solution (W_1, W_2, W_3, W_4) to this system of equations provides a
 4360 solution to the original equation $I_1 = W_1 \cdot W_2 \cdot W_3$ and vice versa. Both equations are therefore
 4361 equivalent in the sense that solutions are in a 1:1 correspondence.

4362 Looking at both equations, we see how each constraint enforces a step in the computation.
 4363 In fact, the first constraint forces any computation to multiply the witness W_1 and W_2 first. Oth-
 4364 erwise it would not be possible to compute the witness W_4 , which is needed to solve the second
 4365 constraint. Witness W_4 therefore expresses the constraining of an intermediate computational
 4366 state.

4367 At this point, one might ask why equation 1 constrains the system to compute $W_1 \cdot W_2$ first,
 4368 since computing $W_2 \cdot W_3$, or $W_1 \cdot W_3$ in the beginning and then multiplying with the remaining
 4369 factor gives the exact same result. The reason is that the way we designed the R1CS prohibits
 4370 any of these alternative computations, which shows that R1CS are in general **not unique** de-
 4371 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

$$\begin{aligned} a_0^1 &= 0 & a_1^1 &= 0 & a_2^1 &= 1 & a_3^1 &= 0 & a_4^1 &= 0 & a_5^1 &= 0 \\ a_0^2 &= 0 & a_1^2 &= 0 & a_2^2 &= 0 & a_3^2 &= 0 & a_4^2 &= 0 & a_5^2 &= 1 \\ b_0^1 &= 0 & b_1^1 &= 0 & b_2^1 &= 0 & b_3^1 &= 1 & b_4^1 &= 0 & b_5^1 &= 0 \\ b_0^2 &= 0 & b_1^2 &= 0 & b_2^2 &= 0 & b_3^2 &= 0 & b_4^2 &= 1 & b_5^2 &= 0 \\ c_0^1 &= 0 & c_1^1 &= 0 & c_2^1 &= 0 & c_3^1 &= 0 & c_4^1 &= 0 & c_5^1 &= 1 \\ c_0^2 &= 0 & c_1^2 &= 1 & c_2^2 &= 0 & c_3^2 &= 0 & c_4^2 &= 0 & c_5^2 &= 0 \end{aligned}$$

With this choice, the rank-1 constraint system of our 3-factorization problem can be written in its most general form as follows:

$$\begin{aligned} (a_0^1 + a_1^1 I_1 + a_2^1 W_1 + a_3^1 W_2 + a_4^1 W_3 + a_5^1 W_4) \cdot (b_0^1 + b_1^1 I_1 + b_2^1 W_1 + b_3^1 W_2 + b_4^1 W_3 + b_5^1 W_4) &= (c_0^1 + c_1^1 I_1 + c_2^1 W_1 + c_3^1 W_2 + c_4^1 W_3 + c_5^1 W_4) \\ (a_0^2 + a_1^2 I_1 + a_2^2 W_2 + a_3^2 W_2 + a_4^2 W_3 + a_5^2 W_4) \cdot (b_0^2 + b_1^2 I_1 + b_2^2 W_2 + b_3^2 W_2 + b_4^2 W_3 + b_5^2 W_4) &= (c_0^2 + c_1^2 I_1 + c_2^2 W_2 + c_3^2 W_2 + c_4^2 W_3 + c_5^2 W_4) \end{aligned}$$

4372 *Example 112* (The Tiny Jubjub curve). Consider the languages $L_{\text{tiny.jj.1}}$ from example 107,
 4373 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 $k = 4$:

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:

$$\begin{aligned}
 (a_0^1 + a_1^1 I_1 + a_2^1 I_2 + a_3^1 W_1 + a_4^1 W_2 + a_5^1 W_3) \cdot (b_0^1 + b_1^1 I_1 + b_2^1 I_2 + b_3^1 W_1 + b_4^1 W_2 + b_5^1 W_3) &= (c_0^1 + c_1^1 I_1 + c_2^1 I_2 + c_3^1 W_1 + c_4^1 W_2 + c_5^1 W_3) \\
 (a_0^2 + a_1^2 I_1 + a_2^2 I_2 + a_3^2 W_1 + a_4^2 W_2 + a_5^2 W_3) \cdot (b_0^2 + b_1^2 I_1 + b_2^2 I_2 + b_3^2 W_1 + b_4^2 W_2 + b_5^2 W_3) &= (c_0^2 + c_1^2 I_1 + c_2^2 I_2 + c_3^2 W_1 + c_4^2 W_2 + c_5^2 W_3) \\
 (a_0^3 + a_1^3 I_1 + a_2^3 I_2 + a_3^3 W_1 + a_4^3 W_2 + a_5^3 W_3) \cdot (b_0^3 + b_1^3 I_1 + b_2^3 I_2 + b_3^3 W_1 + b_4^3 W_2 + b_5^3 W_3) &= (c_0^3 + c_1^3 I_1 + c_2^3 I_2 + c_3^3 W_1 + c_4^3 W_2 + c_5^3 W_3) \\
 (a_0^4 + a_1^4 I_1 + a_2^4 I_2 + a_3^4 W_1 + a_4^4 W_2 + a_5^4 W_3) \cdot (b_0^4 + b_1^4 I_1 + b_2^4 I_2 + b_3^4 W_1 + b_4^4 W_2 + b_5^4 W_3) &= (c_0^4 + c_1^4 I_1 + c_2^4 I_2 + c_3^4 W_1 + c_4^4 W_2 + c_5^4 W_3)
 \end{aligned}$$

4374 In what follows, we write L_{jubjub} for the associated language that consists of solutions to the
 4375 R1CS.

4376 To see that L_{jubjub} is equivalent to $L_{tiny.jj.1}$, let $(I_1, I_2; W_1, W_2, W_3)$ be a word in L_{jubjub} , then
 4377 (I_1, I_2) is a word in $L_{tiny.jj.1}$, since the defining R1CS of L_{jubjub} implies that I_1 and I_2 satisfy the
 4378 Edwards equation of the tiny jubjub curve. On the other hand, let (I_1, I_2) be a word in $L_{tiny.jj.1}$.
 4379 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.

4380 **Exercise 43.** Consider the language $L_{tiny.jj.zk}$ and define a rank-1 constraint relation with a
 4381 decision function such that the associated language is equivalent to $L_{tiny.jj.zk}$.

4382 **R1CS Satisfiability** To understand how rank-1 constraint systems define formal languages,
 4383 observe that every R1CS over a field \mathbb{F} defines a decision function over the alphabet $\Sigma_I \times \Sigma_W =$
 4384 $\mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{R1CS} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{true, false\} ; (I; W) \mapsto \begin{cases} true & (I; W) \text{ satisfies R1CS} \\ false & \text{else} \end{cases} \quad (6.7)$$

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_{jubjub} from example 107 and its associated R1CS. To see how constructive proofs in L_{jubjub} look like, consider the instance $(I_1, I_2) = (11, 6)$. To prove the statement “There exists a witness W such that $(I_1, I_2; W)$ is a word in L_{jubjub} ” constructively, a proof has to provide assignments to all witness variables W_1, W_2 and W_3 . Since the alphabet is \mathbb{F}_{13} , an example assignment is given by $W = (4, 10, 8)$ since $(I_1, I_2; W)$ satisfies the R1CS

$$\begin{array}{ll} I_1 \cdot I_1 = W_1 & 11 \cdot 11 = 4 \\ I_2 \cdot I_2 = W_2 & 6 \cdot 6 = 10 \\ (8 \cdot W_1) \cdot W_2 = W_3 & (8 \cdot 4) \cdot 10 = 8 \\ (12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 = 0 & 12 \cdot 10 + 8 + 10 \cdot 4 + 1 = 0 \end{array}$$

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 135 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:

Definition 6.2.2.1. Algebraic circuit

- The set of edges has a total order.
- Every source node has a label that represents either a variable or a constant from the field \mathbb{F} .

- Every sink node has exactly one incoming edge and a label that represents either a variable or a constant from the field \mathbb{F} .
- Every node that is neither a source nor a sink has exactly two incoming edges and a label from the set $\{+, *\}$ that represents either addition or multiplication in \mathbb{F} .
- All outgoing edges from a node have the same label.
- Outgoing edges from a node with a label that represents a variable have a label.
- Outgoing edges from a node with a label that represents multiplication have a label, if there is at least one labeled edge in both input path.
- All incoming edges to sink nodes have a label.
- If an edge has two labels S_i and S_j it gets a new label $S_i = S_j$.
- No other edge has a label.
- Incoming edges to sink nodes that are labeled with a constant $c \in \mathbb{F}$ are labeled with the same constant. Every other edge label is taken from the set $\{W, I\}$ and indexed compatible with the order of the edge set.

It should be noted that the details in the definitions of algebraic circuits vary between different sources. We use this definition as it is conceptually straightforward and well-suited for pen-and-paper computations.

To get a better intuition of our definition, let $C(\mathbb{F})$ be an algebraic circuit. Source nodes are the inputs to the circuit and either represent variables or constants. In a similar way, sink nodes represent termination points of the circuit and are either output variables or constants. Constant sink nodes enforce computational outputs to take on certain values.

Nodes that are neither source nodes nor sink nodes are called **arithmetic gates**. Arithmetic gates that are decorated with the “+”-label are called **addition-gates** and arithmetic gates that are decorated with the “·”-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

$$f_{3.fac} : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \rightarrow \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 \cdot x_3$$

add reference

check reference

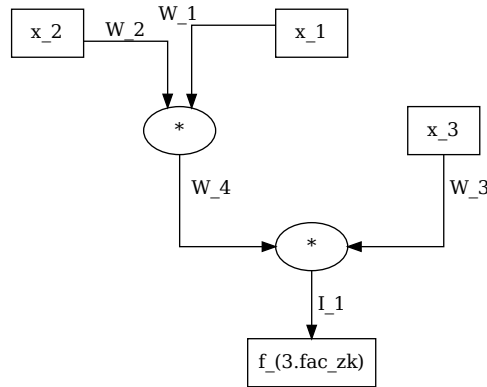
4483 Using this function, we can describe the zero-knowledge 3-factorization problem from 106,
 4484 in the following way: Given instance $I_1 \in \mathbb{F}_{13}$, a valid witness is a preimage of $f_{3, \text{fac}}$ at
 4485 the point I_1 , i.e., a valid witness consists of three values W_1, W_2 and W_3 from \mathbb{F}_{13} such that
 4486 $f_{3, \text{fac}}(W_1, W_2, W_3) = I_1$.

check
reference

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

$$\begin{aligned} f_{3, \text{fac}}(x_1, x_2, x_3) &= x_1 \cdot x_2 \cdot x_3 && \# \text{ bracket choice} \\ &= (x_1 \cdot x_2) \cdot x_3 && \# \text{ operator notation} \\ &= \text{MUL}(\text{MUL}(x_1, x_2), x_3) \end{aligned}$$

4487 Using this expression, we can write an associated algebraic circuit by first constraining the
 4488 variables to edge labels $W_1 = x_1, W_2 = x_2$ and $W_3 = x_3$ as well as $I_1 = f_{3, \text{fac}}(x_1, x_2, x_3)$, taking
 4489 the distinction between private and public inputs into account. We then rewrite the operator
 4490 representation of $f_{3, \text{fac}}$ into circuit nodes and get the following:



4491

4492 In this case, the directed acyclic multi-graph is a binary tree with three leaves (the source
 4493 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
 4494 internal nodes, which are labeled as multiplication gates.

4495 The order we use to label the edges is chosen to make the edge labeling consistent with
 4496 the choice of W_4 as defined in definition 6.2.2.1. This order can be obtained by a depth-first
 4497 right-to-left-first traversal algorithm.

check
reference

Example 116. To give a more realistic example of an algebraic circuit, look at the defining
 equation of the tiny-jubjub curve (66) again. A pair of field elements $(x, y) \in \mathbb{F}_{13}^2$ is a curve
 point, precisely if the following equation holds:

check
reference

$$3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$$

To understand how one might transform this identity into an algebraic circuit, we first rewrite this equation by shifting all terms to the right. We get the following:

$$\begin{aligned} 3 \cdot x^2 + y^2 &= 1 + 8 \cdot x^2 \cdot y^2 && \Leftrightarrow \\ 0 &= 1 + 8 \cdot x^2 \cdot y^2 - 3 \cdot x^2 - y^2 && \Leftrightarrow \\ 0 &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2 \end{aligned}$$

Then we use this expression to define a function such that all points of the tiny-jubjub curve are characterized as the function preimages at 0.

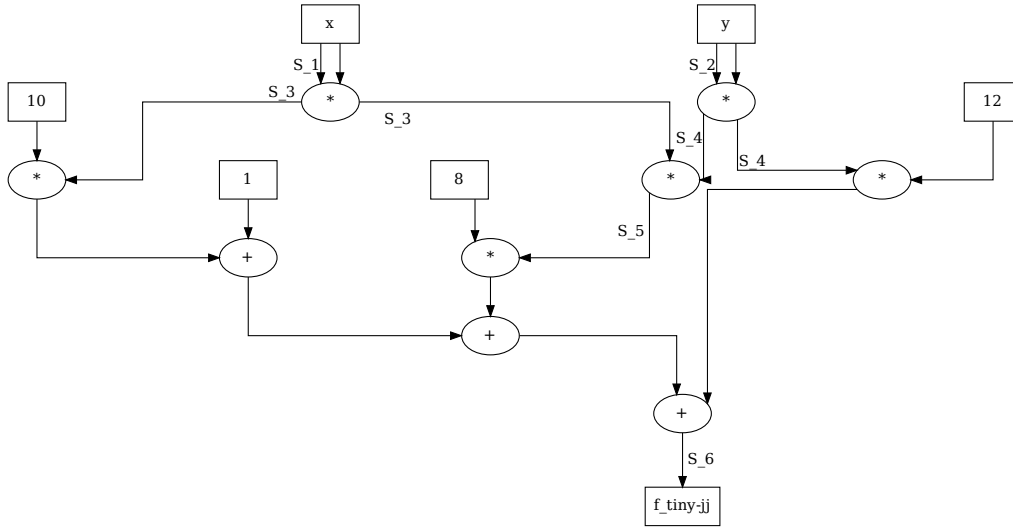
$$f_{\text{tiny-jj}} : \mathbb{F}_{13} \times \mathbb{F}_{13} \rightarrow \mathbb{F}_{13} ; (x, y) \mapsto 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2$$

4498 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
 4499 there are no other curve points. The preimage $f_{\text{tiny-jj}}^{-1}(0)$ is therefore a complete description of
 4500 the tiny-jubjub curve.

We can transform this function into an algebraic circuit over \mathbb{F}_{13} . We first introduce brackets into potentially ambiguous expressions and then rewrite the function in terms of binary operators. We get the following:

$$\begin{aligned} f_{\text{tiny-jj}}(x, y) &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12y^2 && \Leftrightarrow \\ &= ((8 \cdot ((x \cdot x) \cdot (y \cdot y))) + (1 + 10 \cdot (x \cdot x))) + (12 \cdot (y \cdot y)) && \Leftrightarrow \\ &= \text{ADD}(\text{ADD}(\text{MUL}(8, \text{MUL}(\text{MUL}(x, x), \text{MUL}(y, y))), \text{ADD}(1, \text{MUL}(10, \text{MUL}(x, x)))), \text{MUL}(12, \text{MUL}(y, y))) \end{aligned}$$

4501 Since we haven't decided which part of the computation should be public and which part should
 4502 be private, we use the unspecified symbol S to represent edge labels. Constraining all vari-
 4503 ables to edge labels $S_1 = x$, $S_2 = y$ and $S_6 = f_{\text{tiny-jj}}$, we get the following circuit, representing
 4504 the function $f_{\text{tiny-jj}}$, by inductively replacing binary operators with their associated arithmetic
 4505 gates:



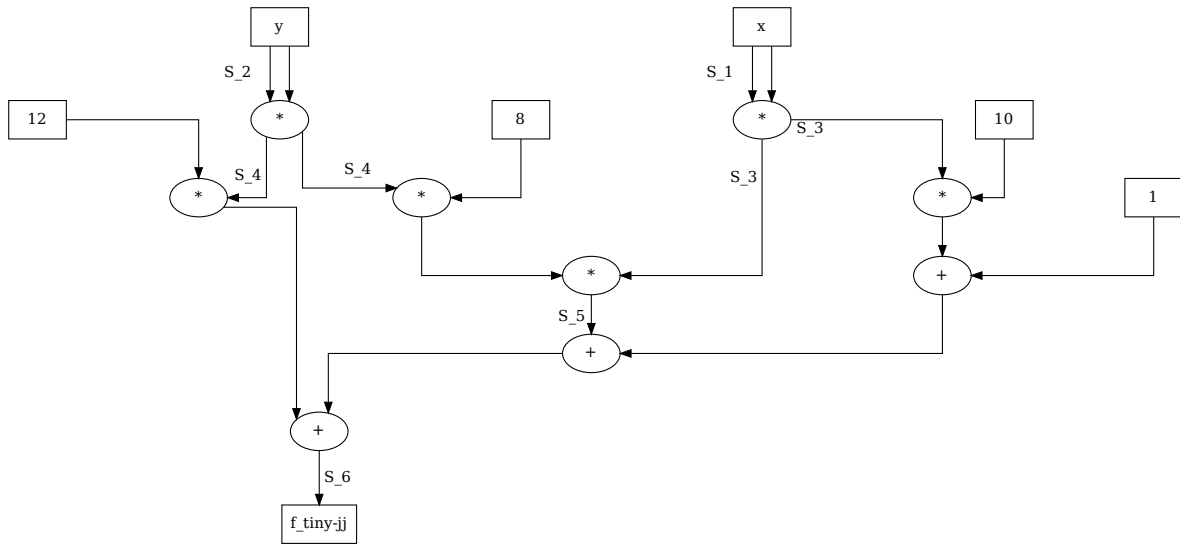
4506

4507 This circuit is not a graph, but a multigraph, since there is more than one edge between some of
 4508 the nodes.

4509 In the process of designing of circuits from functions, it should be noted that circuit rep-
 4510 resentations are not unique in general. In case of the function $f_{\text{tiny-jj}}$, the circuit shape is
 4511 dependent on our choice of bracketing in XXX. An alternative design is, for example, given by
 4512 the following circuit, which occurs when the bracketed expression $8 \cdot ((x \cdot x) \cdot (y \cdot y))$ is replaced
 4513 by the expression $(x \cdot x) \cdot (8 \cdot (y \cdot y))$.

4514

 add refer-
ence



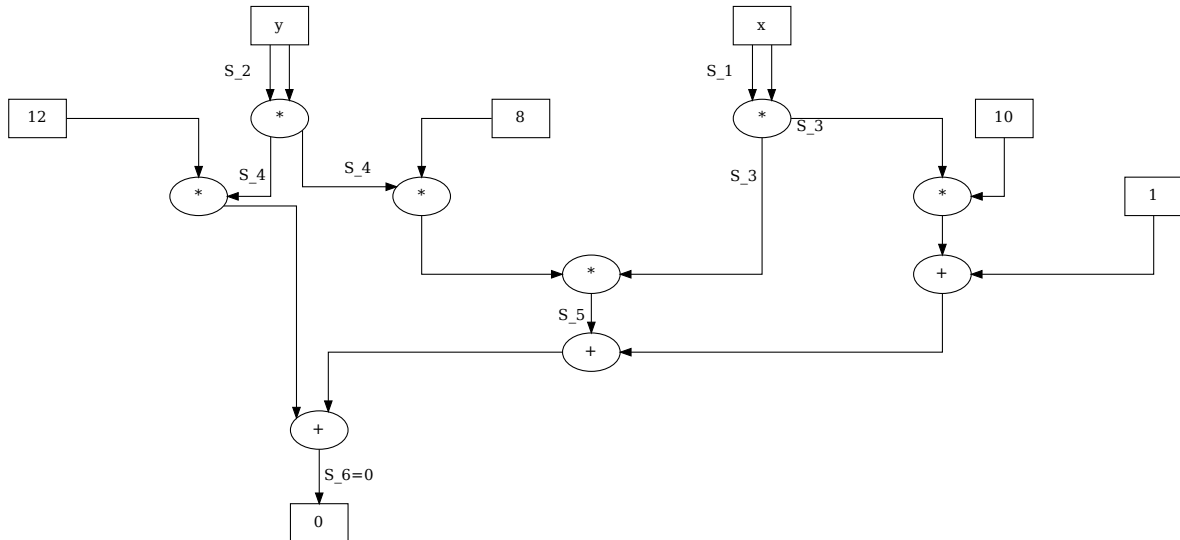
4515

4516

4517 Of course, both circuits represent the same function, due to the associativity and commutativity
 4518 laws that hold true in any field.

4519 With a circuit that represents the function $f_{\text{tiny-jj}}$, we can now proceed to derive a circuit
 4520 that constrains arbitrary pairs (x, y) of field elements to be points on the tiny-jubjub curve. To do
 4521 so, we have to constrain the output to be zero, that is, we have to constrain $S_6 = 0$. To indicate
 4522 this in the circuit, we replace the output variable by the constant 0 and constrain the related edge
 4523 label accordingly. We get the following:

4524

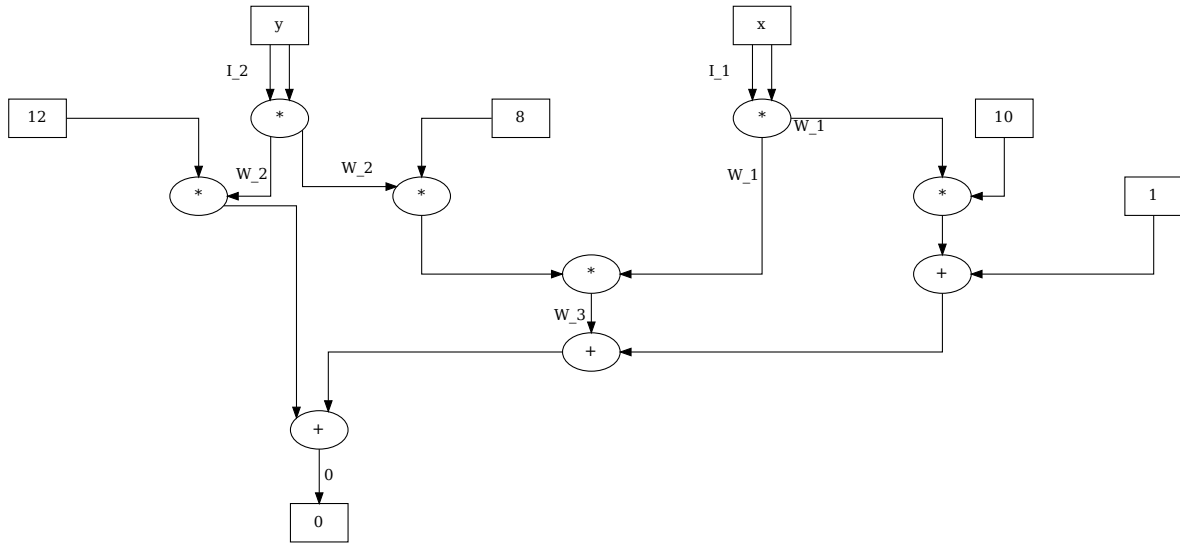


4525

4526

4527 The previous circuit enforces input values assigned to the labels S_1 and S_2 to be points on the
 4528 tiny jubjub curve. However, it does not specify which labels are considered public and which
 4529 are considered private. The following circuit defines the inputs to be public, while all other
 4530 labels are private:

4531



4532

4533

4534 It can be shown that every space- and time-bounded computation can be transformed into
 4535 an algebraic circuit. We call any process that transforms a bounded computation into a circuit
 4536 **flattening**.

4537 **Circuit Execution** Algebraic circuits are directed, acyclic multi-graphs, where nodes repre-
 4538 sent variables, constants, or addition and multiplication gates. In particular, every algebraic
 4539 circuit with n input nodes decorated with variable symbols and m output nodes decorated with
 4540 variables can be seen a function that transforms an input tuple (x_1, \dots, x_n) from \mathbb{F}^n into an out-
 4541 put tuple (f_1, \dots, f_m) from \mathbb{F}^m . The transformation is done by sending values associated to
 4542 nodes along their outgoing edges to other nodes. If those nodes are gates, then the values are
 4543 transformed according to the gate label and the process is repeated along all edges until a sink
 4544 node is reached. We call this computation **circuit execution**.

4545 When executing a circuit, it is possible to not only compute the output values of the circuit
 4546 but to derive field elements for all edges, and, in particular, for all edge labels in the circuit. The
 4547 result is a tuple (S_1, S_2, \dots, S_n) of field elements associated to all labeled edges, which we call a
 4548 **valid assignment** to the circuit. In contrast, any assignment $(S'_1, S'_2, \dots, S'_n)$ of field elements to
 4549 edge labels that can not arise from circuit execution is called an **invalid assignment**.

4550 Valid assignments can be interpreted as **proofs for proper circuit execution** because they
 4551 keep a record of the computational result as well as intermediate computational steps.

4552 *Example 117 (3-factorization).* Consider the 3-factorization problem from example 106 and its
 4553 representation as an algebraic circuit from XXX. We know that the set of edge labels is given
 4554 by $S := \{I_1, W_1, W_2, W_3, W_4\}$.

4555 To understand how this circuit is executed, consider the variables $x_1 = 2$, $x_2 = 3$ as well as
 4556 $x_3 = 4$. Following all edges in the graph, we get the assignments $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$.
 4557 Then the assignments of W_1 and W_2 enter a multiplication gate and the output of the gate is
 4558 $2 \cdot 3 = 6$, which we assign to W_4 , i.e. $W_4 = 6$. The values W_4 and W_3 then enter the second
 4559 multiplication gate and the output of the gate is $6 \cdot 4 = 11$, which we assign to I_1 , i.e. $I_1 = 11$.

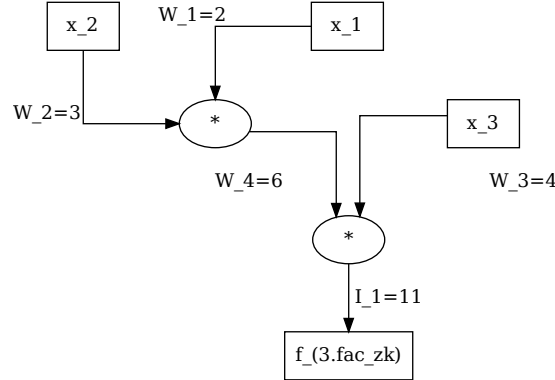
4560 A valid assignment to the 3-factorization circuit $C_{3, fac}(\mathbb{F}_{13})$ is therefore given by the fol-

check
referenceadd refer-
ence

4561 lowing set

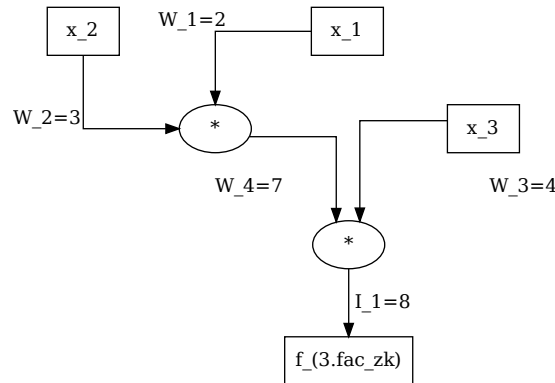
$$S_{\text{valid}} := \{11; 2, 3, 4, 6\} \quad (6.8)$$

4562 We can visualise this assignment in the circuit as follows:



4563

4564 To see what an invalid assignment looks like, consider the assignment $S_{\text{err}} := \{8; 2, 3, 4, 7\}$. In
 4565 this assignment, the input values are the same as in the previous case. The associated circuit is:



4566

4567 This assignment is invalid, as the assignments of I_1 and W_4 cannot be obtained by executing the
 4568 circuit.

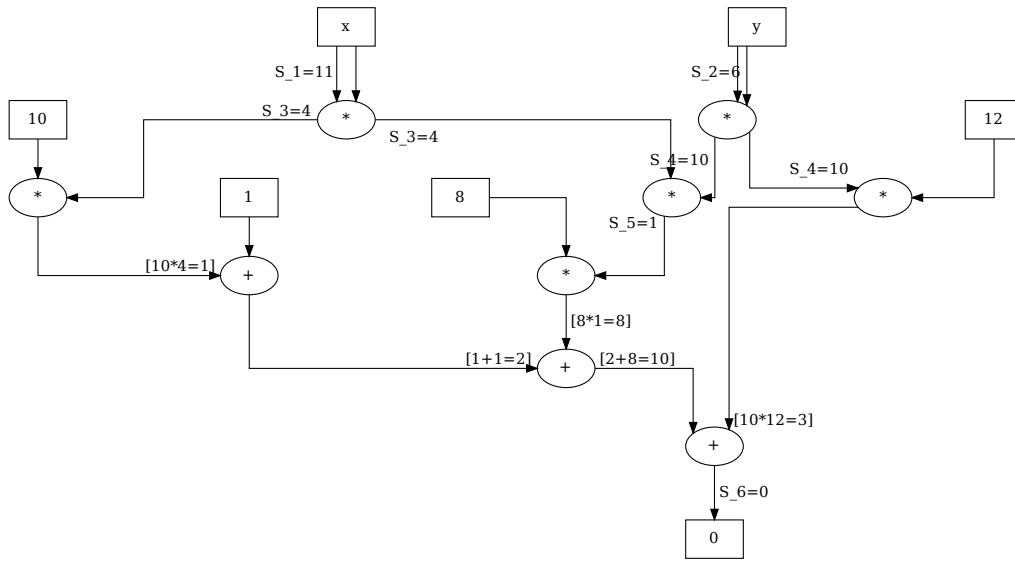
4569 *Example 118.* To compute a more realistic algebraic circuit execution, consider the defining
 4570 circuit $C_{\text{tiny-jj}}(\mathbb{F}_{13})$ from example 114 again. We already know from the way this circuit is
 4571 constructed that any valid assignment with $S_1 = x$, $S_2 = y$ and $S_6 = 0$ will ensure that the pair
 4572 (x, y) is a point on the tiny jubjub curve in its Edwards representation (equation 5.20).

4573 From example 114, we know that the pair $(11, 6)$ is a proper point on the tiny-jubjub curve
 4574 and we use this point as input to a circuit execution. We get the following:

check
reference

check
reference

check
reference



4575

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 the following equation:

$$S_{\text{tiny-jj}} = \{S_1, S_2, S_3, S_4, S_5, S_6\} = \{11, 6, 4, 10, 1, 0\}$$

4576 **Circuit Satisfiability** To understand how algebraic circuits give rise to formal languages, ob-
 4577 serve that every algebraic circuit $C(\mathbb{F})$ over a fields \mathbb{F} defines a decision function over the al-
 4578 phabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{C(\mathbb{F})} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{true, false\} ; (I; W) \mapsto \begin{cases} true & (I; W) \text{ is valid assignment to } C(\mathbb{F}) \\ false & \text{else} \end{cases} \quad (6.9)$$

4579 Every algebraic circuit therefore defines a formal language. The grammar of this language is
 4580 encoded in the shape of the circuit, words are assignments to edge labels that are derived from
 4581 circuit execution, and **statements** are knowledge claims “Given instance I , there is a witness
 4582 W such that $(I; W)$ is a valid assignment to the circuit”. A constructive proof to this claim
 4583 is therefore an assignment of a field element to every witness variable, which is verified by
 4584 executing the circuit to see if the assignment of the execution meets the assignment of the
 4585 proof.

4586 In the context of zero-knowledge proof systems, executing circuits is also often called **wit-**
 4587 **ness generation**, since in applications the instance part is usually public, while its the task of a
 4588 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. 139 here?)

$$L_{\text{CIRCUIT_SAT}(\mathbb{F})} = \{(i; w) \in \Sigma^* \times \Sigma^* \mid \text{there is a circuit } C(\mathbb{F}) \text{ such that } (i; w) \text{ is valid assignment}\}$$

Example 119 (3-Factorization). Consider the circuit $C_{3, fac}$ from equation 6.8 again. We call the associated language L_{3, fac_circ} .

check
reference

To understand how a constructive proof of a statement in L_{3, fac_circ} looks like, consider the instance $I_1 = 11$. To provide a proof for the statement “There exist a witness W such that $(I_1; W)$ is a word in L_{3, fac_circ} ” a proof therefore has to consists of proper values for the variables W_1 , W_2 , W_3 and W_4 . Any proofer therefore has to find input values for W_1 , W_2 and W_3 and then execute the circuit to compute W_4 under the assumption $I_1 = 11$.

Example XXX implies that $(2, 3, 4, 6)$ is a proper constructive proof and in order to verify the proof a verifier needs to execute the circuit with instance $I_1 = 11$ and inputs $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$ to decide whether the proof is a valid assignment or not.

add refer-
ence

Associated Constraint Systems As we have seen in XXX, rank-1 constraint systems define a way to represent statements in terms of a system of quadratic equations over finite fields, suitable for pairing-based zero-knowledge proof systems. However, those equations provide no practical way for a proofer to actually compute a solution. On the other hand, algebraic circuits can be executed in order to derive valid assignments efficiently.

add refer-
ence

In this paragraph, we show how to transform any algebraic circuit into a rank-1 constraint system such that valid circuit assignments are in 1:1 correspondence with solutions to the associated R1CS.

To see this, let $C(\mathbb{F})$ be an algebraic circuit over a finite field \mathbb{F} , with a set of edge labels $\{S_1, S_2, \dots, S_n\}$. Then one of the following steps is executed for every edge label S_j from that set:

- If the edge label S_j is an outgoing edge of a multiplication gate, the R1CS gets a new quadratic constraint

$$(\text{left input}) \cdot (\text{right input}) = S_j \quad (6.10)$$

where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate, and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

- If the edge label S_j is an outgoing edge of an addition gate, the R1CS gets a new quadratic constraint

$$(\text{left input} + \text{right input}) \cdot 1 = S_j \quad (6.11)$$

where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

- No other edge label adds a constraint to the system.

The result of this method is a rank-1 constraint system, and in this sense, every algebraic circuit $C(\mathbb{F})$ generates a R1CS R , which we call the **associated R1CS** of the circuit. It can be shown that a tuple of field elements (S_1, S_2, \dots, S_n) is a valid assignment to a circuit if and only if the same tuple is a solution to the associated R1CS. Circuit executions therefore compute solutions to rank-1 constraints systems efficiently.

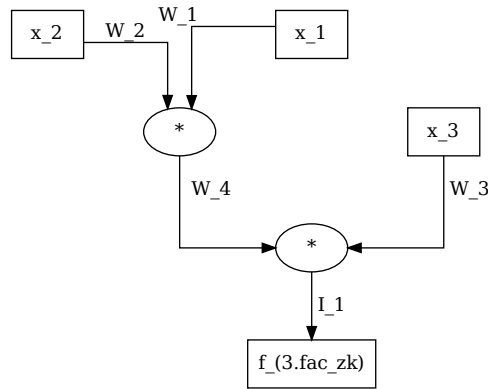
To understand the contribution of algebraic gates to the number of constraints, note that by definition multiplication gates have labels on their outgoing edges if and only if there is at least one labeled edge in both input paths, or if the outgoing edge is an input to a sink node. This

implies that multiplication with a constant is essentially free in the sense that it doesn't add a new constraint to the system, as long as that multiplication gate is not 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 equation 6.8 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.

check
reference



We start with an empty R1CS, and, in order to generate all constraints, we have to iterate over the set of edge labels $\{I_1; W_1, W_2, W_3, W_4\}$.

Starting with the edge label I_1 , we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$\begin{aligned} (\text{left input}) \cdot (\text{right input}) &= I_1 \\ W_4 \cdot W_3 &= I_1 \end{aligned} \quad \Leftrightarrow$$

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 $C_{3, \text{fac}}(\mathbb{F}_{13})$:

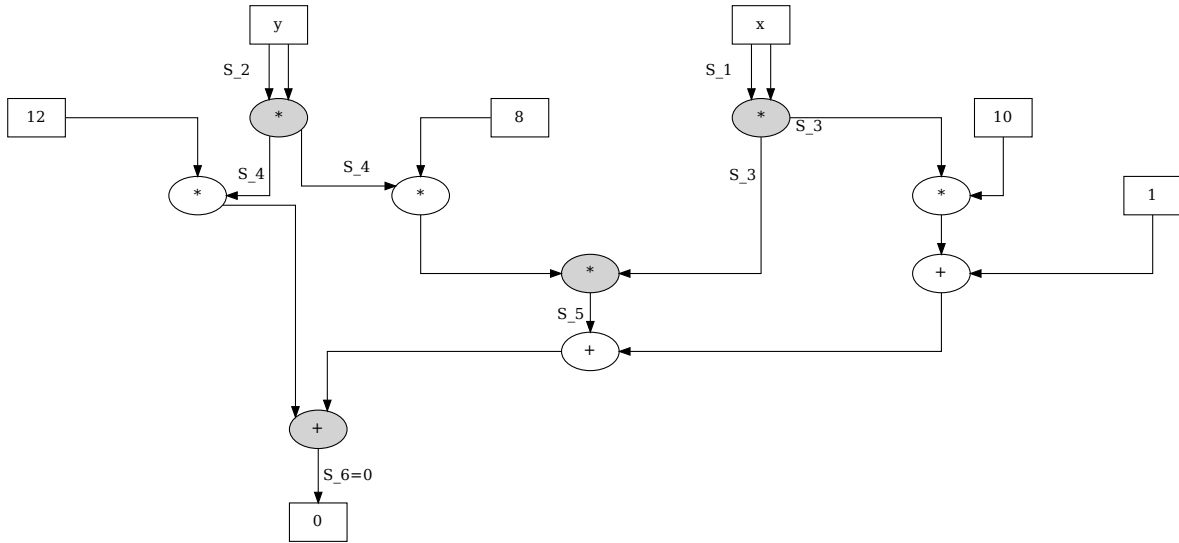
$$\begin{aligned} W_4 \cdot W_3 &= I_1 \\ W_2 \cdot W_1 &= W_4 \end{aligned}$$

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

Example 121. To consider a more general transformation, we consider the tiny-jubjub circuit from example 114 again. A proper circuit is given by

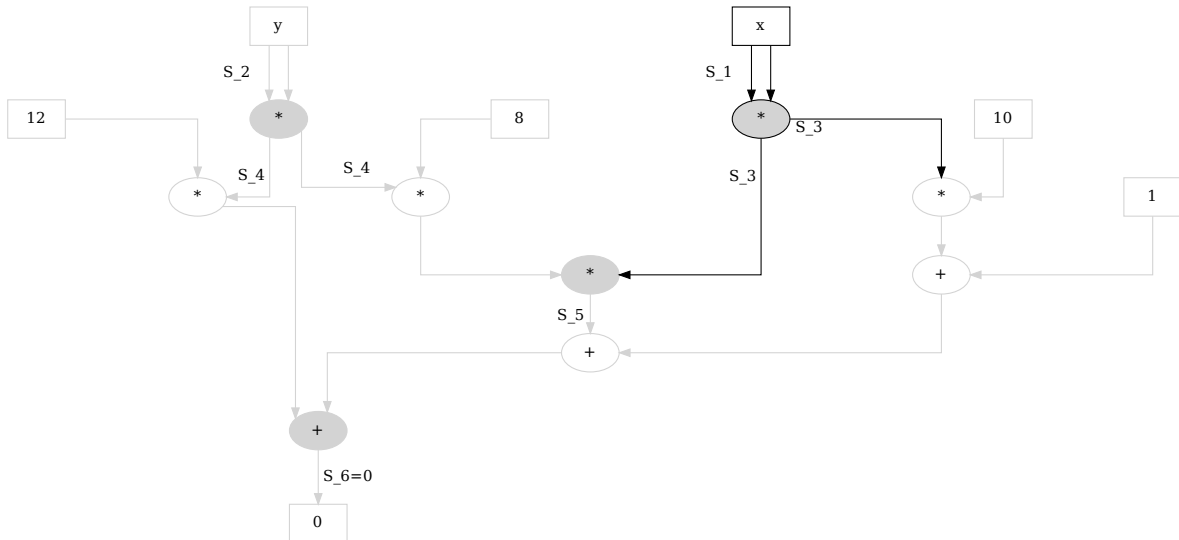
check
reference



To compute the number of constraints, observe that we have 3 multiplication gates that have labels on their outgoing edges and 1 addition gate that has a label on its outgoing edge. We therefore have to compute 4 quadratic constraints.

In order to derive the associated R1CS, we have start with an empty R1CS and then iterate over the set $\{S_1, S_2, S_3, S_4, S_5, S_6 = 0\}$ of all edge labels, in order to generate the constraints.

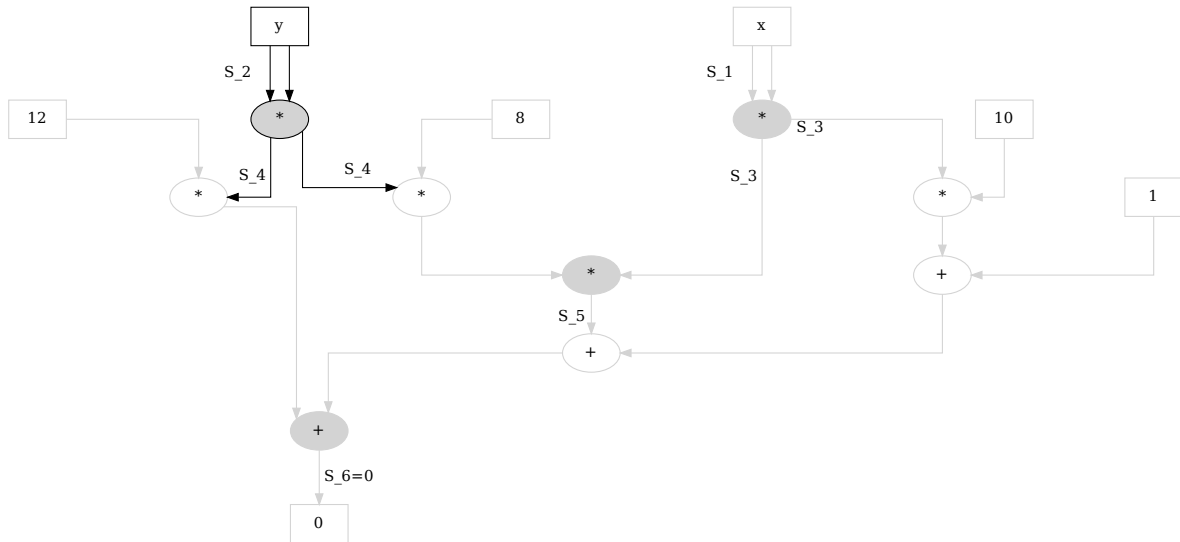
Considering edge label S_1 , we see that the associated edges are not outgoing edges of any algebraic gate, and we therefore have to add no new constraint to the system. The same holds true for edge label S_2 . Looking at edge label S_3 , we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:



Both the left and the right input to this multiplication gate are labeled by S_1 . We therefore have to add the following constraint to the system:

$$S_1 \cdot S_1 = S_3$$

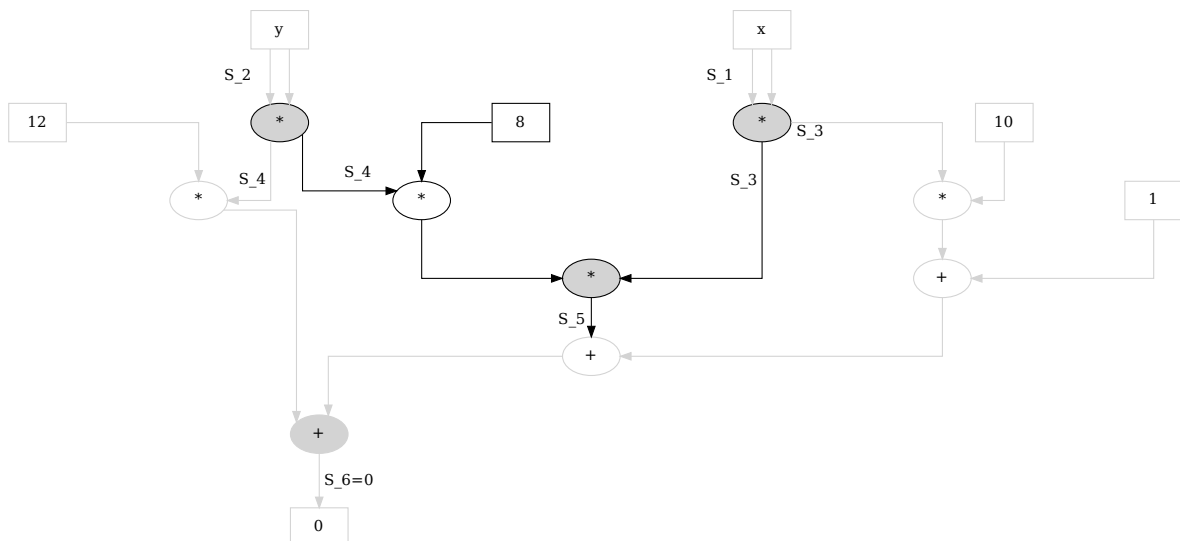
Looking at edge label S_4 , we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:



Both the left and the right input to this multiplication gate are labeled by S_2 and we therefore have to add the following constraint to the system:

$$S_2 \cdot S_2 = S_4$$

Edge label S_5 is more interesting. To see if it implies a constraint, we have to construct the associated subgraph first, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get



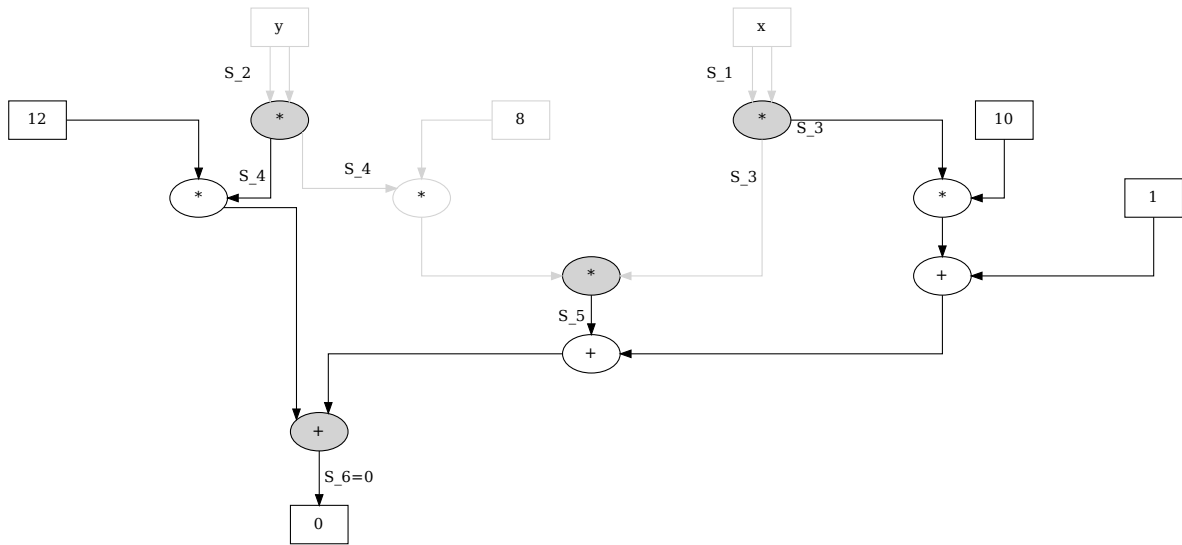
4676

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

$$(S_4 \cdot 8) \cdot S_3 = S_5$$

4677 The last edge label is the constant $S_6 = 0$. To see if it implies a constraint, we have to construct
 4678 the associated subgraph, which consists of all edges and all nodes in all path starting either at a
 4679 constant input or a labeled edge. We get

4680



4681

4682

Both the left and the right input are unlabeled, but have a labeled edges in their path. This implies that we have to add a constraint to the system. Since the gate is an addition gate, the right factor in the quadratic constraint is always 1 and the left factor is computed by symbolically executing all inputs to all gates in sub-circuit. We get

$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

Since there are no more labeled outgoing edges, we are done deriving the constraints. Combining all constraints together, we get the following R1CS:

$$\begin{aligned} S_1 \cdot S_1 &= S_3 \\ S_2 \cdot S_2 &= S_4 \\ (S_4 \cdot 8) \cdot S_3 &= S_5 \\ (12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 &= 0 \end{aligned}$$

4683 which is equivalent to the R1CS we derived in example 114. The languages L_{3, fac_zk} and
 4684 L_{3, fac_circ} are therefore equivalent and both the circuit as well as the R1CS are just two different
 4685 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} :

$$QAP(R) = \left\{ T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{h=0}^{n+m} \right\} \quad (6.12)$$

In the equation above, $T(x) := \prod_{l=1}^k (x - m_l)$ is a polynomial of degree k , called the **target polynomial** of the QAP and A_j, B_j as well as C_j are the unique degree $k - 1$ polynomials defined by the following equation:

$$A_j(m_i) = a_j^i \quad B_j(m_i) = b_j^i \quad C_j(m_i) = c_j^i \quad j = 1, \dots, n + m + 1, i = 1, \dots, k \quad (6.13)$$

Given some rank-1 constraint system, an associated quadratic arithmetic program is therefore nothing but a set of polynomials, computed from the constants in the 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 4 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"?

check 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

$$\begin{aligned} T(x) &= (x - m_1)(x - m_2) && \# \text{ Definition of } T \\ &= (x - 5)(x - 7) && \# \text{ Insert our choice} \\ &= (x + 8)(x + 6) && \# \text{ Negatives in } \mathbb{F}_{13} \\ &= x^2 + x + 9 && \# \text{ expand} \end{aligned}$$

4720 Then we have to compute the polynomials A_j , B_j and C_j by their defining equation from the
4721 R1CS coefficients. Since the R1CS has two constraining equations, those polynomials are of
4722 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:

$$\begin{array}{l} A_0(5) = 0, \quad A_1(5) = 0, \quad A_2(5) = 1, \quad A_3(5) = 0, \quad A_4(5) = 0, \quad A_5(5) = 0 \\ A_0(7) = 0, \quad A_1(7) = 0, \quad A_2(7) = 0, \quad A_3(7) = 0, \quad A_4(7) = 0, \quad A_5(7) = 1 \end{array}$$

$$\begin{array}{l} B_0(5) = 0, \quad B_1(5) = 0, \quad B_2(5) = 0, \quad B_3(5) = 1, \quad B_4(5) = 0, \quad B_5(5) = 0 \\ B_0(7) = 0, \quad B_1(7) = 0, \quad B_2(7) = 0, \quad B_3(7) = 0, \quad B_4(7) = 1, \quad B_5(7) = 0 \end{array}$$

$$\begin{array}{l} C_0(5) = 0, \quad C_1(5) = 0, \quad C_2(5) = 0, \quad C_3(5) = 0, \quad C_4(5) = 0, \quad C_5(5) = 1 \\ C_0(7) = 0, \quad C_1(7) = 1, \quad C_2(7) = 0, \quad C_3(7) = 0, \quad C_4(7) = 0, \quad C_5(7) = 0 \end{array}$$

4723 Lagrange's interpolation implies that a polynomial of degree k , that is, that zero on $k + 1$ points
4724 has to be the zero polynomial. Since our polynomials are of degree 1 and determined on 2
4725 points, we therefore know that the only non-zero polynomials in our QAP are A_2 , A_5 , B_3 , B_4 ,
4726 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:

$$\begin{aligned}
 A_5(x) &= y_0 \cdot l_0 + y_1 \cdot l_1 \\
 &= y_0 \cdot \left(\frac{x - x_1}{x_0 - x_1}\right) + y_1 \cdot \left(\frac{x - x_0}{x_1 - x_0}\right) = 0 \cdot \left(\frac{x - 7}{5 - 7}\right) + 1 \cdot \left(\frac{x - 5}{7 - 5}\right) \\
 &= \frac{x - 5}{2} \quad \# 2^{-1} = 7 \\
 &= 7(x - 5) = 7x + 4 \quad \# -5 = 8 \text{ and } 7 \cdot 8 = 4
 \end{aligned}$$

4727 Using Lagrange's interpolation, we can deduce that $A_2 = B_3 = C_5$ as well as $A_5 = B_4 = C_1$,
 4728 since they are polynomials of degree 1 that evaluate to same values on 2 points. Using this, we
 4729 get the following set of polynomials

$A_0(x) = 0$	$B_0(x) = 0$	$C_0(x) = 0$
$A_1(x) = 0$	$B_1(x) = 0$	$C_1(x) = 7x + 4$
$A_2(x) = 6x + 10$	$B_2(x) = 0$	$C_2(x) = 0$
$A_3(x) = 0$	$B_3(x) = 6x + 10$	$C_3(x) = 0$
$A_4(x) = 0$	$B_4(x) = 7x + 4$	$C_4(x) = 0$
$A_5(x) = 7x + 4$	$B_5(x) = 0$	$C_5(x) = 6x + 10$

4731 We can invoke Sage to verify our computation. In sage every polynomial ring has a function
 4732 `lagrange_polynomial` that takes the defining points as inputs and the associated Lagrange
 4733 polynomial as output.

```

4734 sage: F13 = GF(13)
4735 sage: F13t.<t> = F13[]
4736 sage: T = F13t((t-5)*(t-7))
4737 sage: A2 = F13t.lagrange_polynomial([(5, 1), (7, 0)])
4738 sage: A5 = F13t.lagrange_polynomial([(5, 0), (7, 1)])
4739 sage: T == F13t(t^2 + t + 9)
4740 True
4741 sage: A2 == F13t(6*t + 10)
4742 True
4743 sage: A5 == F13t(7*t + 4)
4744 True

```

4745 Combining this computation with the target polynomial we derived earlier, a quadratic arith-
 4746 metic program associated to the rank-1 constraint system $R_{3.fac_zk}$ is given as follows:

$$\begin{aligned}
 QAP(R_{3.fac_zk}) &= \{x^2 + x + 9, \\
 \{0, 0, 6x + 10, 0, 0, 7x + 4\}, \{0, 0, 0, 6x + 10, 7x + 4, 0\}, \{0, 7x + 4, 0, 0, 0, 6x + 10\}\}
 \end{aligned} \quad (6.14)$$

4747 **QAP Satisfiability** One of the major points of quadratic arithmetic programs in proofing sys-
 4748 tems is that solutions of their associated rank-1 constraints systems are in 1:1 correspondence
 4749 with certain polynomials P such that P is divisible by the target polynomial T of the QAP if and
 4750 only if the solution id a solution. Verifying solutions to the R1CS and hence, checking proper
 4751 circuit execution is then achievable by polynomial division of P by T .

4752 To be more specific, let R be some rank-1 constraints system with associated assignment
 4753 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 :

$$P_{(I;W)} = (A_0 + \sum_j^n I_j \cdot A_j + \sum_j^m W_j \cdot A_{n+j}) \cdot (B_0 + \sum_j^n I_j \cdot B_j + \sum_j^m W_j \cdot B_{n+j}) - (C_0 + \sum_j^n I_j \cdot C_j + \sum_j^m W_j \cdot C_{n+j}) \quad (6.15)$$

Every tuple $(I; W)$ defines a polynomial $P_{(I;W)}$, and, since each polynomial A_j , B_j and C_j is of degree $k-1$, $P_{(I;W)}$ is of degree $(k-1) \cdot (k-1) = k^2 - 2k + 1$.

To understand how quadratic arithmetic programs define formal languages, observe that every QAP over a field \mathbb{F} defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{QAP} : \mathbb{F}^* \times \mathbb{F}^* \rightarrow \{true, false\} ; (I; W) \mapsto \begin{cases} true & P_{(I;W)} \text{ is divisible by } T \\ false & \text{else} \end{cases} \quad (6.16)$$

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 6.2.1.1 and the different ways of computing proofs in both systems. For circuits and their associated rank-1 constraints systems, a constructive proof consists of a valid assignment of field elements to the edges of the circuit, or the variables in the R1CS. However, in the case of QAPs, a valid proof consists of a polynomial $P_{(I;W)}$.

To compute a proof for a statement in L_{QAP} given some instance I , a proofer first needs to compute a constructive proof W , e.g. by executing the circuit. With $(I; W)$ at hand, the proofer can then compute the polynomial $P_{(I;W)}$ and publish it as proof.

Verifying a constructive proof in the case of a circuit is achieved by executing the circuit, comparing the result to the given proof, and verifying the same proof in the R1CS picture means checking if the elements of the proof satisfy all equation.

In contrast, verifying a proof in the case of a QAP is done by polynomial division of the proof P by the target polynomial T of the QAP. The proof checks out if and only if P is divisible by T .

Example 123. Consider the quadratic arithmetic program $QAP(R_{3, fac_zk})$ from example XXX, and its associated R1CS from equation 6.14. 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-

check
reference

add refer-
ence

check
reference

add refer-
ence

lows:

$$\begin{aligned}
P_{(I;W)} &= (A_0 + \sum_j^n I_j \cdot A_j + \sum_j^m W_j \cdot A_{n+j}) \cdot (B_0 + \sum_j^n I_j \cdot B_j + \sum_j^m W_j \cdot B_{n+j}) - (C_0 + \sum_j^n I_j \cdot C_j + \sum_j^m W_j \cdot C_{n+j}) \\
&= (2(6x + 10) + 6(7x + 4)) \cdot (3(6x + 10) + 4(7x + 4)) - (11(7x + 4) + 6(6x + 10)) \\
&= ((12x + 7) + (3x + 11)) \cdot ((5x + 4) + (2x + 3)) - ((12x + 5) + (10x + 8)) \\
&= (2x + 5) \cdot (7x + 7) - (9x) \\
&= (x^2 + 2 \cdot 7x + 5 \cdot 7x + 5 \cdot 7) - (9x) \\
&= (x^2 + x + 9x + 9) - (9x) \\
&= x^2 + x + 9
\end{aligned}$$

4787 Given instance $I_1 = 11$ a proofer therefore provides the polynomial $x^2 + x + 9$ as proof. To verify
4788 this proof, any verifier can then look up the target polynomial T from the QAP and divide $P_{(I;W)}$
4789 by T . In this particular example, $P_{(I;W)}$ is equal to the target polynomial T , and hence, it is
4790 divisible by T with $P/T = 1$. The verification therefore checks the proof.

```

4791 sage: F13 = GF(13) 639
4792 sage: F13t.<t> = F13[] 640
4793 sage: T = F13t(t^2 + t + 9) 641
4794 sage: P = F13t((2*(6*t+10)+6*(7*t+4))*(3*(6*t+10)+4*(7*t+4)) 642
4795             - (11*(7*t+4)+6*(6*t+10)))
4796 sage: P == T 643
4797 True 644
4798 sage: P % T # remainder 645
4799 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 the following remainder:

$$(8x^2 + 6)/(x^2 + x + 9) = 8 + \frac{5x + 12}{x^2 + x + 9}$$

4800 This implies that $P_{(I;W)}$ is not divisible by T , and hence, the verification does not check the
4801 proof. Any verifier can therefore show that the proof is false.

```

4802 sage: F13 = GF(13) 647
4803 sage: F13t.<t> = F13[] 648
4804 sage: T = F13t(t^2 + t + 9) 649
4805 sage: P = F13t((2*(6*t+10)+8*(7*t+4))*(3*(6*t+10)+4*(7*t+4)) - ( 650
4806             8*(6*t+10)+11*(7*t+4)))

```

4807	<code>sage: P == F13t(8*t^2 + 6)</code>	651
4808	<code>True</code>	652
4809	<code>sage: P % T # remainder</code>	653
4810	<code>5*t + 12</code>	654

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>.
- P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.
- 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>.