1 Zk-Snark

The following is a collection of materials regarding this hardly believable technology. Some subjects are objectively complex, but even though 'white papers' are necessary, sometimes they seem to be a narcissistic exercise of beautiful complex formulas, that nobody except the writers can understand. Technologies are sometimes complex and not for everyone, but there is too often a lack of easier documentation with simple examples to understand the low level details. Something that should be good for security reasons but more in general to help everyone to understand the technologies. One of the problems of the crypto world, in my opinion, is that they are too hard and complex to understand for most of the people, and probably this is why there are still people who think that Bitcoin is bullshit, because "there's nothing behind it". There's a lack of efforts to produce easier and understandable documentation.

Chapter 2 is an introduction to this topic with a few simple examples on what is **zk-snark**, and why it could be useful. This is something that everybody could read and understand, it only takes 5 minutes or so.

Chapter 3 is an explanation to better understand the post from Vitalik Buterin, which lacks some steps to understand the topic in more detail. This is where I have put most of my efforts to better clarify things.

https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649

The most important and complete explanation on the subject are from this guy, who made a wonderful job:

https://twitter.com/maksympetkus

https://arxiv.org/pdf/1906.07221.pdf

https://medium.com/@imolfar/why-and-how-zk-snark-works-1-introduction-the-medium-of-a-proof-d946e931160

The 8 posts have been detailed in chapters 5-10, they have been simply copied and pasted.

Another very well done site, full of interesting high level material, is the following:

https://zkproof.org/2020/10/15/information-theoretic-proof-systems-part-ii/

This doc is long and full of many potentially boring details, in case the math behind this technology is not of your interest, just stop after the first pages that are a general introduction to the problem and the infinite areas of its applicability.

"ZK STARK" is an evolution of the zk-snark algorithm, it's simpler since it doesn't use elliptic curves and homomorphic concepts, and uses quantum proof hashes to produce computation proofs, which are slightly longer than those produced with zk-snark algorithm. When proofs need to be stored on public block chains systems, this should be taken into consideration. Moreover, they scale better with the problem's complexity, and they don't need the so called 'cerimony' for the non-interactive case, thus it could be considered more secure, since no trap doors that could compromise the system exist. The balance is between security, the length of the produced proof, interactivity between the prover and the verifier.

https://starkware.co/stark/

Some more details could be added about the above topic in the future, trying to demystify a bit the following (in my opinion still not trivial) explanations:

https://vitalik.ca/general/2017/11/09/starks_part_1.html

https://vitalik.ca/general/2017/11/22/starks_part_2.html

https://vitalik.ca/general/2018/07/21/starks_part_3.html

In any case, understanding zk-snark is useful anyway also tohelp you understading zk-stark, since many concepts about polynomials are the same.

2 Introduction

The concept of 'proof of computation' is surprisingly old and there is a paper published in 1989 by "Shafi Goldwasser, Silvio Micali, and Charles Rackoff". Silvio Micali is the same one, the founder of **Algorand**, a public proof of steak blockchain system.
https://epubs.siam.org/doi/10.1137/0218012

Due to the rising of blockchain technologies, these kind of techniques, adding privacy features, have become more and more important for a couple of reasons:

- adding privacy to public blockchain systems, could make these systems available and attractive also for private companies
- offloading computations and saving just the small proofs that complex computations have been correctly performed, helps scaling the most used layer1 blockchains like Ethereum, which start getting overloaded (with a substantial increase in gas fees). These techniques are usually referred to as "zk rollups".

Zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARK*) is the truly ingenious method of proving that something is true without revealing any other information, however, why it is useful in the first place? *Zero-knowledge proofs* are advantageous in a myriad of application, including:

1) Proving statement on private data:
- Person *A* has more than *X* in his bank account
- In the last year, a bank did not transact with an entity *Y*
- Matching DNA without revealing full DNA
- One has a credit score higher than *Z*

2) Anonymous authorization:
- Proving that requester *R* has right to access web-site's restricted area without revealing its identity (e.g., login, password)
- Prove that one is from the list of allowed countries/states without revealing from which one exactly
- Prove that one owns a monthly pass to a subway/metro without revealing card's id

3) Anonymous payments:
- Payment with full detachment from any kind of identity [Ben+14]
- Paying taxes without revealing one's earnings

4) Outsourcing computation:
- outsource an expensive computation and validate that the result is correct without redoing the execution; it opens up a category of trustless computing
- changing a blockchain model from everyone computes the same to one party computes and everyone verifies

- *scale the TPS value by performing computations on 'parachains' or secondary chains, and putting on a main chain only the necessary data (e.g. a merkle root hash), that can be verified in an easy, fast and cheap way with a smart contract*

As great as it sounds on the surface the underlying method is a "marvel" of mathematics and cryptography and is being researched for the 4th decade since its introduction in 1985 in the principal work "The Knowledge Complexity of Interactive Proof-systems" [GMR85] with subsequent introduction of the non-interactive proofs [BFM88] which are especially essential in the context of blockchains.
In any *zero-knowledge proof* system, there is a *prover* who wants to convince a *verifier* that some *statement* is true without revealing any other information, e.g., *verifier* learns that the *prover* has more than *X* in his bank account but nothing else (i.e., the actual amount is not disclosed). A protocol should satisfy three properties:

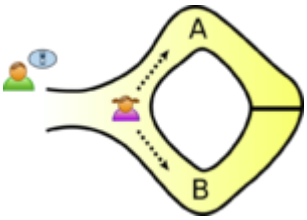
- Completeness — if the *statement* is true then a (honest) *prover* can convince a *verifier*
- Soundness — a cheating *prover* can not convince a *verifier* of a false *statement*
- Zero-knowledge — the interaction only reveals if a *statement* is true and nothing else

The *zk-SNARK* term itself was introduced in [Bit+11], building on [Gro10] with following Pinocchio protocol [Gen+12; Par+13] making it applicable for general computing. Pinocchio is a character of the Italian fairy tales writer Carlo Collodi: Pinocchio is a wooden child whose nose grows up when he lies. Follow hereafter some other examples of 'zero knowledge interactive proofs'.

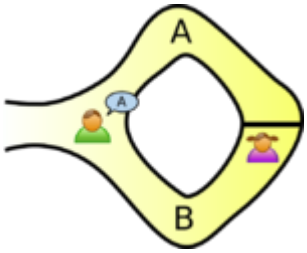
2.1 Ali Baba Cave

https://en.wikipedia.org/wiki/Zero-knowledge_proof

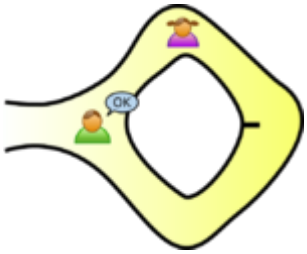
Victor wants to be sure that Peggy has the secret code to open the door at the end of the cave. How can Peggy demonstrate this to Victor, without actually giving him the key to try himself to open the door ? This would be the interactive zero knowledge algorithm (or protocol):



Peggy randomly takes either path A or B, while Victor waits outside (i.e. Victor doesn't know if Peggy has chosen path A or B).



Victor chooses an **exit** path.



Peggy reliably appears at the exit Victor names.

Of course, Peggy could have just been lucky, because Victor has chosen the same exit that Peggy decided to enter at the beginning. But repeating this many times, for example 100 times, there are two possibilities:

1. Peggy comes back from the wrong side for more or less 50% of the times
2. Peggy come always back from the right side that Victor chose

In the latter case, Peggy has the key to open the cave's door, otherwise she doesn't have it.

2.2 Colored balls

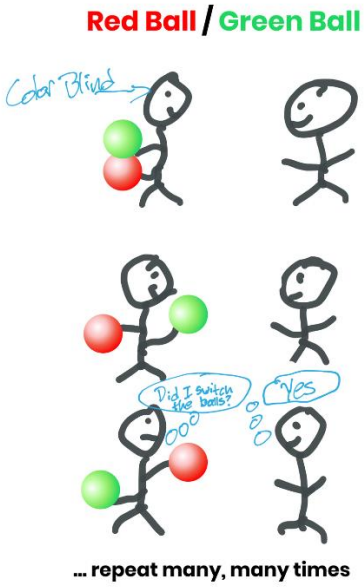
https://www.notboring.co/p/zero-knowledge

Bob is color blind and can't say if the two balls he holds are of the same color or not. Mark claims he can demonstrate that to Bob, without telling Bob the color of the balls. The protocol is the following one:

Bob hides the balls behind his back, and decides to switch them or not. Than shows them to Mark, who has to say if Bob switched the balls in his hands or not. If the game is repeated 100 times, averagely speaking Mark:

- will answer in the right way 100% of the times if the two balls are of different colors
- will answer in the right way 50% of the times if the two balls are of the same color

In any case, supposing Mark is trusted and doesn't lie, Bob will know if the two balls are of the same color or not.



2.3 Leaves on the tree

Victor has some kind of super powers, so he can count the number of leaves on a tree. Alice doesn't believe him (of course not in this case), so she wants to be sure about it, but Victor doesn't want to reveal the total number of leaves on the tree. So Victor doesn't look to Alice when se removes N leaves (not all of them

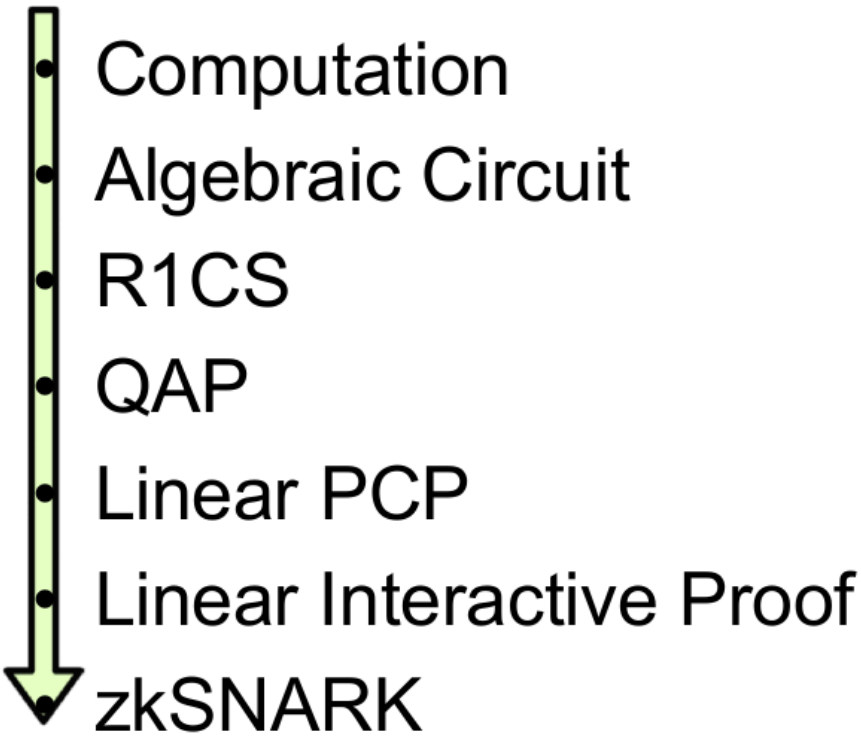
obviously, just a few of them …) from the tree. Victor looks again at the tree, and tells to Alice how many leaves she has stolen from the tree. In this case we probably don't need to repeat the experiment many times.

3 Vitalik Buterin example with R1cs 'circuits'

At the following link you can find some explanations from Vitalik Buterin, Ethereum's co-founder together with Gavin Wood. We copy hereafter the same example, adding some more details that have been omitted and prevent from fully understand the topic. Moreover, it introduces the concept of R1CS which is anyway useful in this context.

https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649

The purpose of this post is not to serve as a full introduction to zk-SNARKs; it assumes as background knowledge that (i) you know what zk-SNARKs are and what they do, and (ii) know enough math to be able to reason about things like polynomials (if the statement $P(x) + Q(x) = (P + Q)(x)$, where P and Q are polynomials, seems natural and obvious to you, then you're at the right level). Rather, the post digs deeper into the machinery behind the technology, and tries to explain as well as possible the first half of the pipeline, as drawn by zk-SNARK researcher Eran Tromer here:



The steps here can be broken up into two halves. First, zk-SNARKs cannot be applied to any computational problem directly; rather, you have to convert the problem into the right "form" for the problem to operate on. The form is called a "quadratic arithmetic program" (QAP), and transforming the code of a function into one of these is itself highly nontrivial. Along with the process for converting the code of a function into a QAP is another process that can be run alongside so that if you have an input to the code you can create a corresponding solution (sometimes called "witness" to the QAP). After this, there is another fairly intricate process for creating the actual "zero knowledge proof" for this witness, and a separate process for verifying a proof that someone else passes along to you, but these are details that are out of scope for this post.
The example that we will choose is a simple one: proving that you know the solution to a cubic equation:

$x^3 + x + 5 = 35$

This problem is simple enough that the resulting QAP will not be so large as to be intimidating, but nontrivial enough that you can see all of the machinery come into play. Let us write out our function as follows:

```
def qeval(x):
    y = x**3
    return x + y + 5
```

The simple special-purpose programming language that we are using here supports basic arithmetic (+, -, *, /), constant-power exponentiation (x^7 but not x^y) and variable assignment, which is powerful enough that you can theoretically do any computation inside of it (as long as the number of computational steps is bounded; no loops allowed). Note that modulo (%) and comparison operators (<, >, ≤, ≥) are NOT supported, as there is no efficient way to do modulo or comparison directly in finite cyclic group arithmetic (be thankful for this; if there was a way to do either one, then elliptic curve cryptography would be broken faster than you can say "binary search" and "Chinese remainder theorem").

You can extend the language to modulo and comparisons by providing bit decompositions (eg. 13 = 2^3 + 2^2+ 1) as auxiliary inputs, proving correctness of those decompositions and doing the math in binary circuits; in finite field arithmetic, doing equality (==) checks is also doable and in fact a bit easier, but these are both details we won't get into right now. We can extend the language to support conditionals (eg. if x < 5: y = 7; else: y = 9) by converting them to an arithmetic form: y = 7 * (x < 5) + 9 * (x >= 5); though note that both "paths" of the conditional would need to be executed, and if you have many nested conditionals then this can lead to a large amount of overhead.

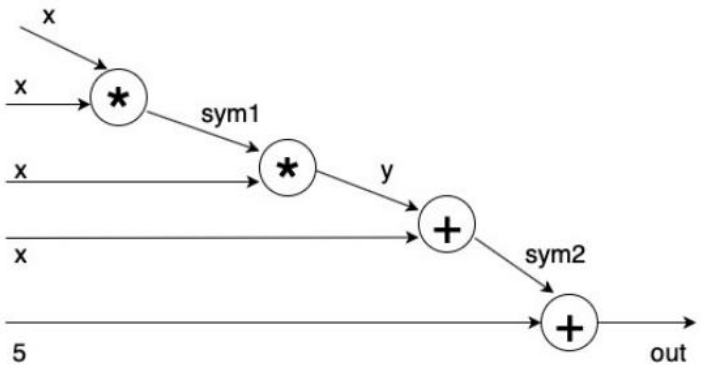
Let us now go through this process step by step. If you want to do this yourself for any piece of code, I implemented a compiler here (for educational purposes only; not ready for making QAPs for real-world zk-SNARKs quite yet!).

3.1 Flattening

The first step is a "flattening" procedure, where we convert the original code, which may contain arbitrarily complex statements and expressions, into a sequence of statements that are of two forms:x = y (where y can be a variable or a number)and x = y (op) z (where op can be +, -, *, / and y and z can be variables, numbers or themselves sub-expressions). You can think of each of these statements as being kind of like logic gates in a circuit. The result of the flattening process for the above code is as follows:

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

If you read the original code and the code here, you can fairly easily see that the two are equivalent. Hereafter is represented a "circuit scheme" that perform the SAME above operations:



As ‘x’ changes as an input variable, the output variable changes as well. The equation corresponding to this circuit is supposed to be known to the prover and the verifier, while the verifier needs to ensure, given a certain output, that the computation has been correctly performed. The ‘zero knowledge’ will be added encrypting values in the proper way, but in this example we just want to show how the algorithm works. Many complex equations can be translated happily into a R1CS circuit, in real life applications there could potentially be millions of parameters and gates. The calculation of a hash value could by itself imply thousands of gates. For example the following piece of code:

```
function calc(w, a, b)
 if w then
   return a × b
 else
   return a + b
 end if
end function
```

could be represented as follows, with the input variable ‘w’ used to decide which is the output:

$$f(w,a,b) = w(a \times b) + (1-w)(a+b)$$

See also the following site:
https://zokrates.github.io/introduction.html

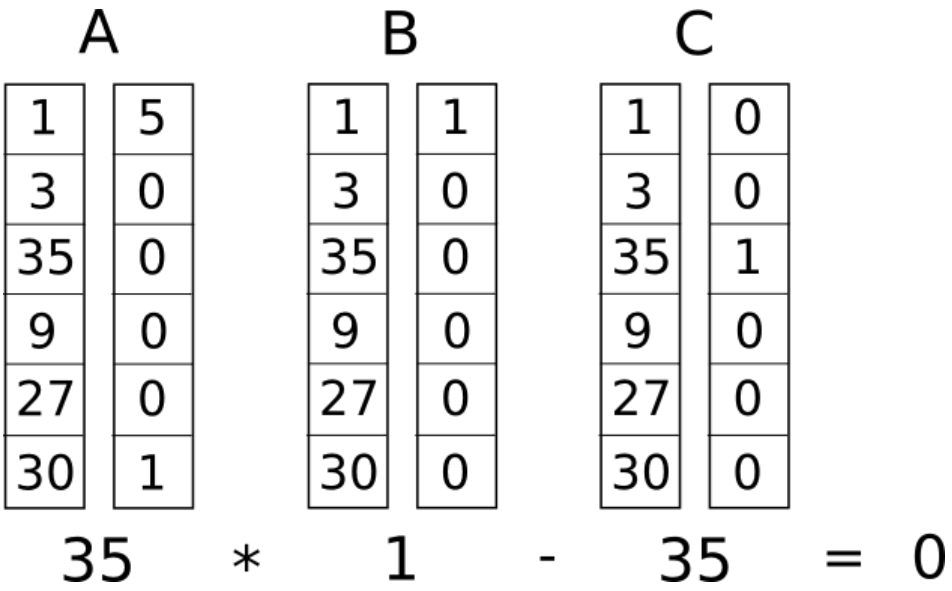
The above circuit is somewhat similar to neural networks (the gates being somewhat like the neurons), except for the fact that everything is deterministic here, while on neural networks you change x and try to find the middle layer parameters to minimize the observed output y. Beware that the above circuit is valid for ANY x and y (or output), so that once you have solved it, you can re-use it for ANY possible input. This will be clear after having read the rest of the document.

3.2 Gates to R1CS

Now, we convert this into something called a rank-1 constraint system (R1CS). An R1CS is a sequence of groups of three vectors (a, b, c), and the solution to an R1CS is a vector s, where s must satisfy the equation:

$$s \cdot a * s \cdot b - s \cdot c = 0$$

where . represents the dot product - in simpler terms, if we "zip together" a and s, multiplying the two values in the same positions, and then take the sum of these products, then do the same to b and s and then c and s, then the third result equals the product of the first two results. For example, this is a satisfied R1CS:



A few more important details here: the above is NOT a multiplication of matrixes, since dimensions are clearly WRONG. A multiplication between a matrix A of N rows and M columns and a matrix B of M rows (yes, the row number of B MUST be equal to the column's number of A) and P columns is a matrix C of N rows and P columns. The vector S (also called witness) is that of the variables we have used in the circuit, adding '1' to represent constants:

['1', 'x', 'out', 'sym_1', 'y', 'sym_2']

Let's first consider the first gate, representing the equation:

sym_1 = x * x

x * x - sym_1 = 0

The equation:

s . a * s . b - s . c = 0

or equivalently:

a. s * b . s - c . s = 0

for the first gate becomes:

$$\left([0,1,0,0,0,0] * \begin{bmatrix} 1 \\ x \\ out \\ sym\_1 \\ y \\ sym\_2 \end{bmatrix} \right) * \left([0,1,0,0,0,0] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} \right) - [0,0,0,1,0,0] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} = 0$$

Considering just one single gate, this representation works fine even with matrixes multiplication. But adding the subsequent gates, this is not true anymore. For the sake of space and for a more 'elegant' and compact representation, we will keep a matrix representation, just keep in mind that the '*' does NOT represent a matrix multiplication, but rather a LINE by LINE multiplication. Let's consider the second gate:

y = sym_1 * x

sym_1 * x – y = 0

$$\left([\,0,0,0,1,0,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym\_1 \\ y \\ sym\_2 \end{bmatrix} \right) * \left([\,0,1,0,0,0,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} \right) - [\,0,0,0,0,1,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} = 0$$

Third gate:

sym_2 = y + x
(y + x)*1 – sym_2 = 0

$$\left([\,0,1,0,0,1,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym\_1 \\ y \\ sym\_2 \end{bmatrix} \right) * \left([\,1,0,0,0,0,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} \right) - [\,0,0,0,0,1,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} = 0$$

Fourth gate:

out = sym_2 + 5
(sym_2 + 5)*1 – out = 0

$$\left([\,5,0,0,0,0,1\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym\_1 \\ y \\ sym\_2 \end{bmatrix} \right) * \left([1,0,0,0,0,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} \right) - [\,0,0,1,0,0,0\,] * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} = 0$$

Providing a compact representation similar to:

a. s * b . s - c . s = 0

we obtain the following:

$$\left(\begin{bmatrix} 0,1,0,0,0,0 \\ 0,0,0,1,0,0 \\ 0,1,0,0,1,0 \\ 5,0,0,0,0,1 \end{bmatrix} * \begin{bmatrix} 1 \\ x \\ out \\ sym\_1 \\ y \\ sym\_2 \end{bmatrix} \right) * \left(\begin{bmatrix} 0,1,0,0,0,0 \\ 0,1,0,0,0,0 \\ 1,0,0,0,0,0 \\ 1,0,0,0,0,0 \end{bmatrix} * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} \right) - \begin{bmatrix} 0,0,0,1,0,0 \\ 0,0,0,0,1,0 \\ 0,0,0,0,1,0 \\ 0,0,1,0,0,0 \end{bmatrix} * \begin{bmatrix} 1 \\ x \\ out \\ sym_1 \\ y \\ sym_2 \end{bmatrix} = 0$$

As already said, **don't confuse the '*' symbol with the matrix multiplication** (the values are instead simply multiplied **row by row**). The **witness** is simply the assignment to all the variables, including input, output and internal variables:

```
[ '1','x','~out','sym_1','y','sym_2'] → [1, 3, 35, 9, 27, 30]
```

You can compute this for yourself by simply "executing" the flattened code above, starting off with the input variable assignment x=3, and putting in the values of all the intermediate variables and the output as you compute them. The above equation becomes:

$$\left(\begin{bmatrix}0,1,0,0,0,0\\0,0,0,1,0,0\\0,1,0,0,1,0\\5,0,0,0,0,1\end{bmatrix}*\begin{bmatrix}1\\3\\35\\9\\27\\30\end{bmatrix}\right) * \left(\begin{bmatrix}0,1,0,0,0,0\\0,1,0,0,0,0\\1,0,0,0,0,0\\1,0,0,0,0,0\end{bmatrix}*\begin{bmatrix}1\\3\\35\\9\\27\\30\end{bmatrix}\right) - \begin{bmatrix}0,0,0,1,0,0\\0,0,0,0,1,0\\0,0,0,0,0,1\\0,0,1,0,0,0\end{bmatrix}*\begin{bmatrix}1\\3\\35\\9\\27\\30\end{bmatrix} = 0$$

$$\left(\begin{bmatrix}3\\9\\3+27\\5+30\end{bmatrix}\right) * \left(\begin{bmatrix}3\\3\\1\\1\end{bmatrix}\right) - \begin{bmatrix}9\\27\\30\\35\end{bmatrix} = 0$$

where the last equation stands considering the multiplication '*' of the two matrixes as a line by line multiplication.

3.3 R1CS to QAP

The next step is taking this R1CS and converting it into QAP (quadratic arithmetic program) form, which implements the exact same logic except using polynomials instead of dot products.

$$A = \begin{bmatrix}0,1,0,0,0,0\\0,0,0,1,0,0\\0,1,0,0,1,0\\5,0,0,0,0,1\end{bmatrix} \quad B = \begin{bmatrix}0,1,0,0,0,0\\0,1,0,0,0,0\\1,0,0,0,0,0\\1,0,0,0,0,0\end{bmatrix} \quad C = \begin{bmatrix}0,0,0,1,0,0\\0,0,0,0,1,0\\0,0,0,0,0,1\\0,0,1,0,0,0\end{bmatrix}$$

We go from four groups of three vectors of length six, to six groups of 3 degree polynomials, where evaluating the polynomials at each *x coordinate* represents one of the constraints. Let's consider the first column of matrix A, the constraints are the following:

$$A_1(x=1) = 0$$
$$A_1(x=2) = 0$$
$$A_1(x=3) = 0$$
$$A_1(x=4) = 5$$

Beware that the 4 'x' values can be arbitrarily chosen, but they MUST be the same for all polynomials, to have a valid equation. Since the number of gates is equal to the polynomial's degree (i.e. 4 in this example), we need to find the following coefficients:

$$a_3x^3 + a_2x^2 + a_1x^1 + a_0$$

… to respect to above values for x=1, x=2, x=3, x=4.
Let's consider the second column of matrix A, the constraints are the following:

$$A_2(x=1) = 1$$
$$A_2(x=2) = 0$$
$$A_2(x=3) = 1$$
$$A_2(x=4) = 0$$

Since we have imposed 4 constraints for a polynomial of degree 3, there is just one solution for such a polynomial (this is intuitive but can be demonstrated). The same can be done for columns 3,4,5,6 of matrix A, and for matrixes B and C for a total of 18 polynomials.

We can make this transformation using something called a *Lagrange interpolation*. The problem that a Lagrange interpolation solves is this: if you have a set of points (ie. (x, y) coordinate pairs), then doing a Lagrange interpolation on those points gives you a polynomial that passes through all of those points. We do this by decomposing the problem: for each x coordinate, we create a polynomial that has the desired y coordinate at that x coordinate and a y coordinate of 0 at all the other x coordinates we are interested in, and then to get the final result we add all of the polynomials together. For example, for the first polynomial we have the following 4 equations:

$(1)\ a_3(x=1)^3+a_2(x=1)^2+a_1(x=1)^1+\ a_0=0$
$(1)\ a_3+a_2+a_1+a_0=0$

$(2)\ a_3(x=2)^3+a_2(x=2)^2+a_1(x=2)^1+\ a_0=0$
$(2)\ 8a_3+4a_2+2a_1+a_0=0$

$(3)\ a_3(x=3)^3+a_2(x=3)^2+a_1(x=3)^1+\ a_0=0$
$(3)\ 27a_3+9a_2+3a_1+a_0=0$

$(4)\ a_3(x=4)^3+a_2(x=4)^2+a_1(x=4)^1+\ a_0=0$
$(4)\ 64a_3+16a_2+4a_1+a_0=0$

The above system of equations have a unique solution, you solve equation (1) for a specific parameter, for example:

$(1)\ a_3+a_2+a_1=-a_0$

Replace the above parameter in the other 3 equations, thus obtaining 3 linear equations with 3 parameters. Repeat the process to find out the exact values of the 4 coefficients.

With exactly the coordinates that we want. The algorithm as described above takes O(n^3) time, as there are n points and each point requires O(n^2) time to multiply the polynomials together; with a little thinking, this can be reduced to O(n^2) time, and with a lot more thinking, using fast Fourier transform algorithms and the like, it can be reduced even further — a crucial optimization when functions that get used in zk-SNARKs in practice often have many thousands of gates.

Now, let's use Lagrange interpolation to transform our R1CS. What we are going to do is take the first value out of every a vector, use Lagrange interpolation to make a polynomial out of that (where evaluating the polynomial at i gets you the first value of the ith a vector), repeat the process for the first value of every b and c vector, and then repeat that process for the second values, the third, values, and so on. For convenience I'll provide the answers right now:

$$A=\begin{bmatrix} -5 & 8 & 0 & -6 & 4 & -1 \\ 9.166 & -11.333 & 0 & 9.5 & -7 & 1.833 \\ -5 & 5 & 0 & -4 & 3.5 & -1 \\ 0.833 & -0.666 & 0 & 0.5 & -0.5 & 0.166 \end{bmatrix}$$

Coefficients are in ascending order, so the first polynomial above is actually 0.833 * x^3 — 5*x^2+ 9.166*x - 5.

$$B=\begin{bmatrix} 3 & -2.0 & 0 & 0 & 0 & 0 \\ -5.166 & 5.166 & 0 & 0 & 0 & 0 \\ 2.5 & -2.5 & 0 & 0 & 0 & 0 \\ -0.333 & 0.333 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & -1 & 4 & -6 & 4.0 \\ 0 & 0 & 1.833 & -4.333 & 9.5 & -7 \\ 0 & 0 & -1 & 1.5 & -4 & 3.5 \\ 0 & 0 & 0.166 & -0.166 & 0.5 & -0.5 \end{bmatrix}$$

This set of polynomials (plus a Z polynomial that I will explain later) makes up the parameters for this particular QAP instance. Note that all of the work up until this point needs to be done only once for every function that you are trying to use zk-SNARKs to verify; once the QAP parameters are generated, they can be reused (*of course, to solve the same problem but with different inputs*).
Let's try evaluating all of these polynomials at x=1. Evaluating a polynomial at x=1 simply means adding up all the coefficients (as 1**k = 1 for all k), so it's not difficult. We get:

```
A results at x=1
0
1
0
0
0
0

B results at x=1
0
1
0
0
0
0

C results at x=1
0
0
0
1
0
0
```

And lo and behold, what we have here is exactly the same as the set of three vectors for the first logic gate that we created above.

From the following site:
https://devdocs.platon.network/docs/en/Verifiable_Computation/

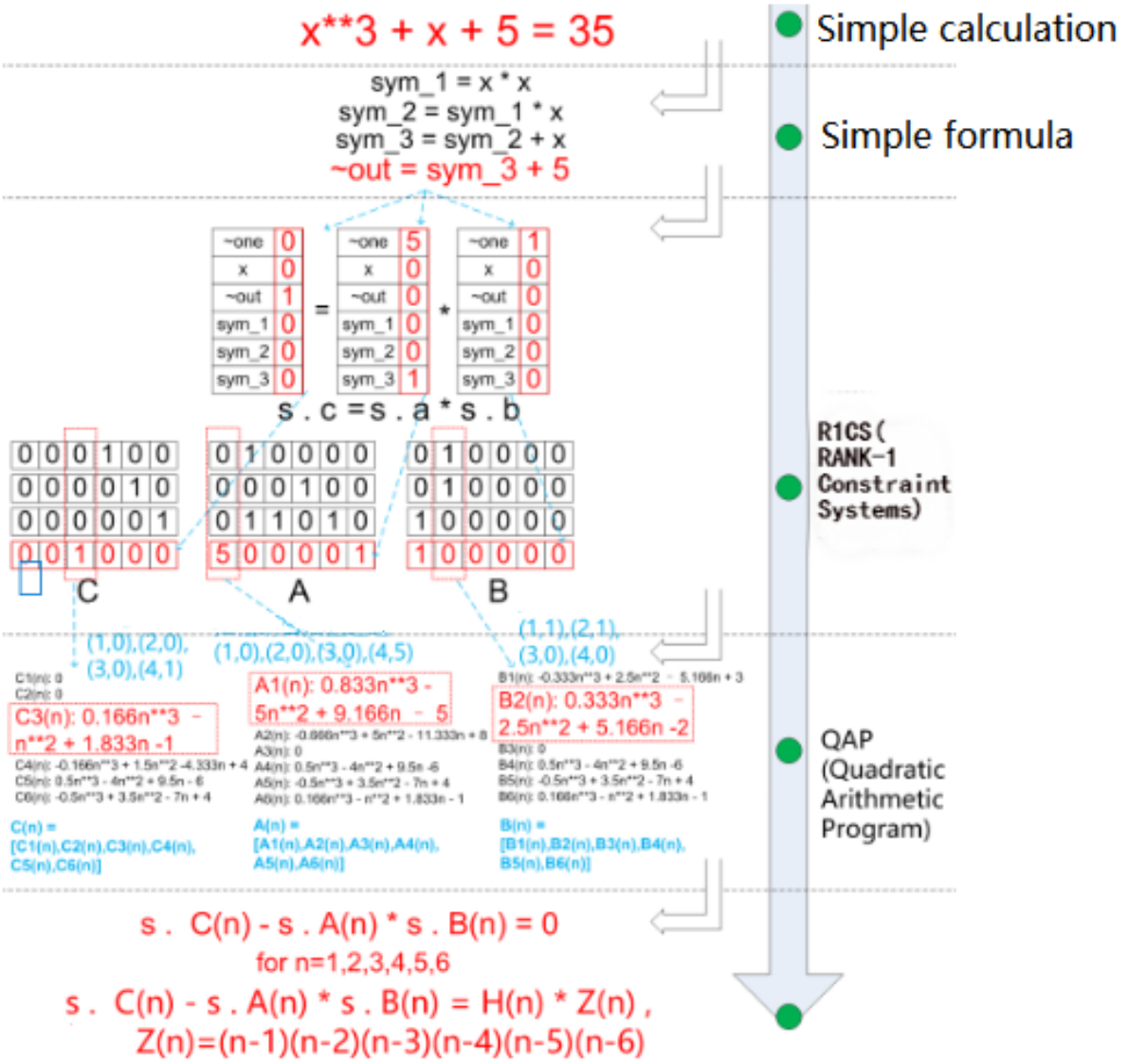
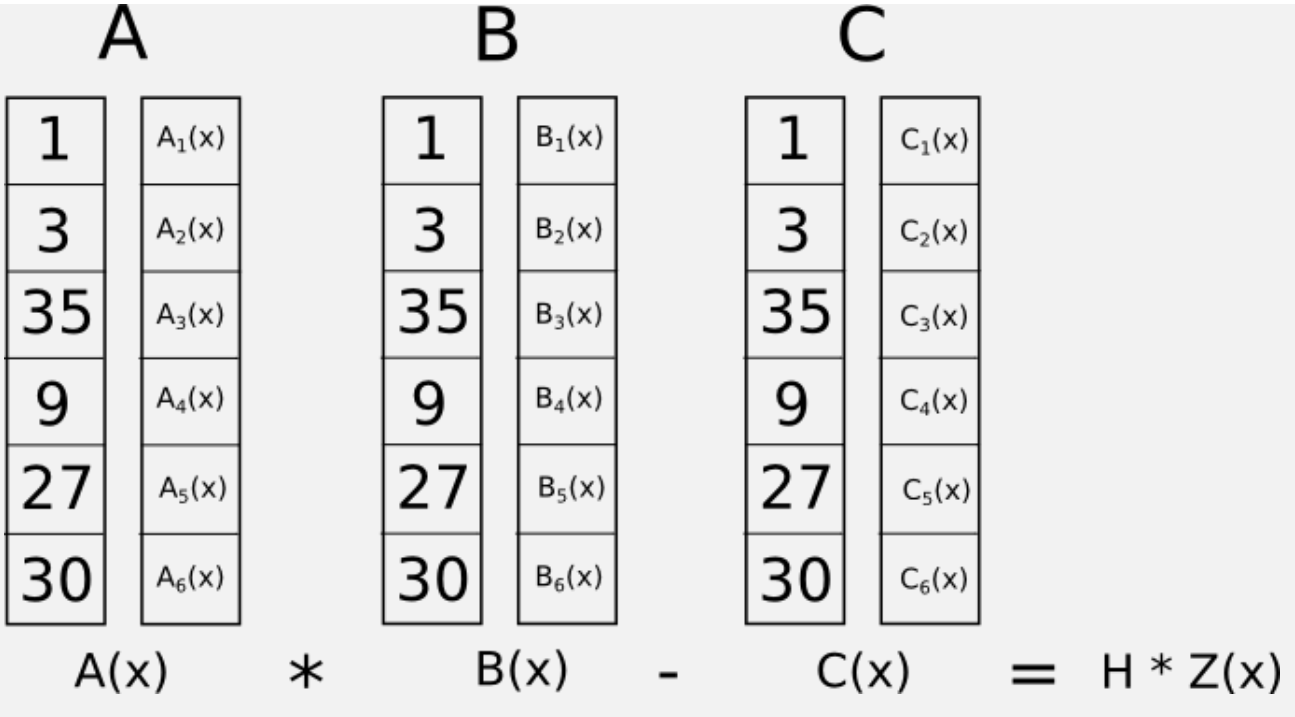


Figure 1: Transforming the computing equation to QAP

The above picture seems to be clear about how values are used and correlated, but it's probably wrong to write that Z(n) also has a zero for n=5 and n=6. Z(n) has zeroes on n=1,2,3,4 while H(n) has to be calculated dividing the left side of the equation for Z(n) (no remainder is expected), but it doesn't necessarily have zeroes for n=5 and n=6.

3.4 Checking the QAP

Now what's the point of this crazy transformation? The answer is that instead of checking the constraints in the R1CS individually, we can now check *all of the constraints at the same time* by doing the dot product check *on the polynomials*.



Because in this case the dot product check is a series of additions and multiplications of polynomials, the result is itself going to be a polynomial. If the resulting polynomial, evaluated at every x coordinate that we used above to represent a logic gate, is equal to zero, then that means that all of the checks pass; if the resulting polynomial evaluated at at least one of the x coordinate representing a logic gate gives a nonzero value, then that means that the values going into and out of that logic gate are inconsistent (ie. the gate is y = x * sym_1 but the provided values might be x = 2,sym_1 = 2 and y = 5).

Note that the resulting polynomial does not itself have to be zero, and in fact in most cases won't be; it could have any behavior at the points that don't correspond to any logic gates, as long as the result is zero at all the points that *do* correspond to some gate. To check correctness, we don't actually evaluate the polynomial:

t = A . s * B . s - C . s

… at every point corresponding to a gate; instead, we divide t by another polynomial, Z, and check that Z evenly divides t - that is, the division t / Z leaves no remainder.

Z is defined as (x - 1) * (x - 2) * (x - 3) ... - the simplest polynomial that is equal to zero at all points that correspond to logic gates. It is an elementary fact of algebra that *any* polynomial that is equal to zero at all of these points has to be a multiple of this minimal polynomial, and if a polynomial is a multiple of Z then its evaluation at any of those points will be zero; this equivalence makes our job much easier. Now, let's actually do the dot product check with the polynomials above. First, the intermediate polynomials:

$$A.s = \begin{bmatrix} -5 & 8 & 0 & -6 & 4 & -1 \\ 9.166 \cdot x & -11.333 \cdot x & 0 & 9.5 \cdot x & -7 \cdot x & 1.833 \cdot x \\ -5 \cdot x^2 & 5 \cdot x^2 & 0 & -4 \cdot x^2 & 3.5 \cdot x^2 & -1 \cdot x^2 \\ 0.833 \cdot x^3 & -0.666 \cdot x^3 & 0 & 0.5 \cdot x^3 & -0.5 \cdot x^3 & 0.166 \cdot x^3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 35 \\ 9 \\ 27 \\ 30 \end{bmatrix} =$$

$$\begin{bmatrix} -5 + 24 - 64 + 108 - 30 \\ (9.166 - 34 + 85.5 - 189 + 55) \cdot x \\ (-5 + 15 - 36 + 94.5 - 30) \cdot x^2 \\ (0.833 - 2 + 4.5 - 13.5 + 5) \cdot x^3 \end{bmatrix} = \begin{bmatrix} 43 \\ -73.333 \cdot x \\ 38.5 \cdot x^2 \\ -5.166 \cdot x^3 \end{bmatrix}$$

Also beware that if we sum up the above 18 polynomials that we retrieved through matrix A, matrix B and matrix C, the equation ***A . s * B . s - C . s*** becomes the following one:

$(A_1(x) \cdot 1 + A_2(x) \cdot 3 + A_3(x) \cdot 35 + A_4(x) \cdot 9 + A_5(x) \cdot 27 + A_6(x) \cdot 30) * (B_1(x) \cdot 1 + B_2(x) \cdot 3 + B_3(x) \cdot 35 + B4_3(x) \cdot 9 + B_5(x) \cdot 27 + B_6(x) \cdot 30) - (C_1(x) \cdot 1 + C_2(x) \cdot 3 + C_3(x) \cdot 35 + C_5(x) \cdot 9 + C_6(x) \cdot 27 + A_6(x) \cdot 30)$

This is a generic polynomial that can be solved for any x, but its value is exactly 0 for x=1, 2, 3, 4 (or any 4 values of 'x' we chose, in the world of numbers we're working) : this is because we have found all the coefficients of the 18 polynomials to respect the initial equation.
In the same way we obtain:

$$B.s = \begin{bmatrix} -3 \\ 10.333 \cdot x \\ -5 \cdot x^2 \\ 0.666 \cdot x^3 \end{bmatrix}$$

$$C.s = \begin{bmatrix} -41 \\ 71.666 \cdot x \\ -24.5 \cdot x^2 \\ 2.833 \cdot x^3 \end{bmatrix}$$

Now we need to calculate (again the multiplication A.s with B.s has NOTHING to do with matrix multiplication, this can create quite a lot of confusion …):

A . s * B . s — C . s = t(x)

t(x) = (-5.166· x^3 + 38.5· x^2 - 73.333· $x + 43) \cdot (-0.666 \cdot x^3 - 5 \cdot x^2 + 10.333 \cdot x - 3) - (2.833 \cdot x^3 - 24.5 \cdot x^2 + 71.666 \cdot x - 41)$

t = [-88.0, 592.666, -1063.777, 805.833, -294.777, 51.5, -3.444]
t (x) = -3.444· x^6 + 51.5· x^5 – 294.777· $x^4 + 805.833 \cdot x^3 - 1063.777 \cdot x^2 + 592.666 \cdot x - 88$

Since t(x) has zeroes for x=1,2,3,4 this means it can be written as:

t(x)= H(x) · Z(x)

Now let's calculate the polynomial Z(x):
Z = (x - 1) * (x - 2) * (x - 3) * (x - 4):
Z = [24, -50, 35, -10, 1]
Z = x^4 - 10· x^3 + 35· $x^2 - 50 \cdot x + 24$

Since the minimal polynomial t(x) for the way the constraints has been imposed must have zeroes for x=1, x=2, x=3 and x=4, this means it can be written as:

t(x) = H(x) · Z(x)

If we divide the result above by Z(x), we get:

H(x) = $\frac{t(x)}{Z(x)}$ = [-3.666, 17.055, -3.444] = -3.444· $x^2 + 17.055 \cdot x - 3.666$

(As we have previously noted, H(x) doesn't have zeroes for x=5 and x=6)

With no remainder (since t(x) MUST be divisible by Z(x), since they have been built in this way).

And so we have the solution for the QAP. If we try to falsify any of the variables in the R1CS solution that we are deriving this QAP solution from — say, set the last one to 31 instead of 30, then we get a t polynomial that fails one of the checks (in that particular case, the result at x=3 would equal -1 instead of 0), and furthermore t would not be a multiple of Z; rather, dividing t / Z would give a remainder of [-5.0, 8.833, -4.5, 0.666].
Note that the above is a simplification; "in the real world", the addition, multiplication, subtraction and division will happen not with regular numbers, but rather with **finite field elements** — a spooky kind of arithmetic which is self-consistent, so all the algebraic laws we know and love still hold true, but where all answers are elements of some finite-sized set, usually integers within the range from 0 to n-1 for some n. For example, if n = 13, then 1 / 2 = 7 (and 7 * 2 = 1), 3 * 5 = 2, and so forth. Using finite field arithmetic removes the need to worry about rounding errors and allows the system to work nicely with elliptic curves, which end up being necessary for the rest of the zk-SNARK machinery that makes the zk-SNARK protocol actually secure.

3.5 Summary

Now let's recap what we have achieved now:

1- a way to transform a computer program into a circuit, using libraries or other techniques
2- the circuit is 'transformed' into a polynomial, which when evaluated on a certain number of x values, is equal to zero and this has the consequence that input values, output values and intermediate values of the calculation that the circuit represents, are correct with no doubts

So if the prover and the verifier BOTH know this polynomial, the verifier could choose a certain random value 'x', and ask to the prover what is the value 'y' of such a polynomial for such a random value. What is the probability that the prover is just lucky and answers with the right value, if he doesn't know the polynomial ? it is intuitively true that two polynomials with degree N can only have the same values (i.e. intersect each other) on N points, in any case it can be proven. For example two lines are a polynomial of degree 1, they can be parallel or cross in just one point. If you can choose x in a space of 10^{77} numbers, such probability would be negligible: N/10^{77}. Choosing one value of 'x' instead of all values provides the concept of 'succinctness' (producing a proof can take a lot of time, but verification is extremely fast). This example is just to have an idea of where we're going to, and how zero knowledge proofs are related to such arguments like r1cs circuits, QAP. Privacy is related to perform calculations and checks in another 'world', that of the encrypted values. We'll go in more detail afterwards.

3.6 Working in a finite field world of numbers Zp

Working with real numbers is not the best thing to do, equal is equal and rounding errors are not safe nor secure for any protocol or algorithm (coefficients like 0.333 are not good). Rounding errors are one of the most frequent errors when working with 'Solidity' and smart contracts, where tokens, Ether, gas are all represented by integer values.
For this reason, in real life applications we always work on the world of finite Z_p numbers, 'p' being a prime number and numbers being integers going from 0 to (p-1). Let's take as an example p=11, just to understand how things work, but keep in mind that in real life applications p is a very large number made for example of 100 digits.

$$Z_{11}, x \in [0,1,2,3,4,5,6,7,8,9,10]$$

In this world, all the usual rules we know apply, with the only difference that all numbers 'wrap' around the highest possible value p, 11 in this case. So every number $x_{mod\ p}$ is the rest of the integer division of x for the prime number 'p'. Additions and multiplications can be performed in the way we already know, with the usual rules all valid. For example with p=11:

$$(5+9)_{mod\ p} = 14_{mod\ p} = 3$$

$(5+15)_{mod\ p} = 20_{mod\ p} = 9$
$(13+15)_{mod\ p} = 13_{mod\ p} + 15_{mod\ p} = 2_{mod\ p} + 4_{mod\ p} = 6_{mod\ p}$
$(13+15)_{mod\ p} = 28_{mod\ p} = 6_{mod\ p}$

$(5 \cdot 9)_{mod\ p} = 45_{mod\ p} = 1$
$(5 \cdot 15)_{mod\ p} = 75_{mod\ p} = 9$
$(13 \cdot 15)_{mod\ p} = 13_{mod\ p} \cdot 15_{mod\ p} = 2_{mod\ p} \cdot 4_{mod\ p} = 8_{mod\ p}$
$(13 \cdot 15)_{mod\ p} = 195_{mod\ p} = (17 \cdot 11 + 8)_{mod\ p} = 8_{mod\ p}$

We can also define the division operation in the following intuitive way. For example suppose we want to find the inverse of 2 in the world Z_{11}:
$2^{-1} = \frac{1}{2}$

The inverse of 2 is a number $x\ \epsilon\ Z_{11}$, such that:

$(x \cdot 2)_{mod\ p} = 1$

We can easily find such a number since p is very small:

$(6 \cdot 2)_{mod\ p} = 12_{mod\ p} = 1$

In the same way, we can find the inverse number for all the values belonging to Z_{11}:

$1^{-1}_{mod\ p} = 1, \quad 2^{-1}_{mod\ p} = 6, \quad 3^{-1}_{mod\ p} = 4, \quad 4^{-1}_{mod\ p} = 3, \quad 5^{-1}_{mod\ p} = 9$

$6^{-1}_{mod\ p} = 2, \quad 7^{-1}_{mod\ p} = 8, \quad 8^{-1}_{mod\ p} = 7, \quad 9^{-1}_{mod\ p} = 5, \quad 10^{-1}_{mod\ p} = 10$

What about the minus operation ? suppose we want to find the negative of 2 in the world Z_{11}, this is a number x such that:

$(-2 + x)_{mod\ p} = 0$

We can easily find such a number since p is very small:

$(2+9)_{mod\ p} = 11_{mod\ p} = 0$

In the same way, we can find the inverse number for all the values belonging to Z_{11}:

$-1_{mod\ p} = 10, -2_{mod\ p} = 9, \quad -3_{mod\ p} = 8, \quad -4_{mod\ p} = 7, \quad -5_{mod\ p} = 6$
$-6_{mod\ p} = 5, \quad -7_{mod\ p} = 4, \quad -8_{mod\ p} = 3, \quad -9_{mod\ p} = 2, \quad -10_{mod\ p} = 1$

We now have all the necessary rules and prerequisites to apply the same rules to the equations we have previously defined in the 'real' world of numbers $\mathbb{R}$. As you can see in the following site:

https://asecuritysite.com/encryption/go_qap

… the example we have previously considered is used to provide the polynomial t(x) output in the world of Z_{11}. You can see how the flattening has more gates than necessary, since:

x (Private input): 3, y (Public input): 35
r (Prime): 11
Flat: func exp3(private a):
    b = a * a
    c = a * b
    return c

func main(private s0, public s1):
    s3 = exp3(s0)
    s4 = s3 + s0
    s5 = s4 + 5
    equals(s1, s5)
    out = 1 * 1 ← this one is completely useless

… so the witness has 8 values (=variables) instead of just 6, adding unnecessary complexity to the solution. Such optimizations are very important when you work with thousands or millions of gates. Considering again the first polynomial of matrix A but in the finite world of numbers Z_{11}:

$A_1(x=1)=0$
$A_1(x=2)=0$
$A_1(x=3)=0$
$A_1(x=4)=5$

$a_3x^3+a_2x^2+a_1x^1+\ a_0$ with $x, a_3, a_2, a_1, a_0 \in Z_{11}$

The system of 4 equations, remembering that we're working on Z_{11}, becomes the following:

(1) $a_3(x=1)^3+a_2(x=1)^2+a_1(x=1)^1+\ a_0=0$
(1) $a_3+a_2+a_1+a_0=0$

(2) $a_3(x=2)^3+a_2(x=2)^2+a_1(x=2)^1+\ a_0=0$
(2) $8a_3+4a_2+2a_1+a_0=0$

(3) $a_3(x=3)^3+a_2(x=3)^2+a_1(x=3)^1+\ a_0=0$
(3) $27a_3+9a_2+3a_1+a_0=5a_3+9a_2+3a_1+a_0=0$

(4) $a_3(x=4)^3+a_2(x=4)^2+a_1(x=4)^1+\ a_0=5$
(4) $64a_3+16a_2+4a_1+a_0=9a_3+5a_2+4a_1+a_0=5$

The 4 equations become:
(1) $a_0=-a_3-a_2-a_1=10a_3+10a_2+10a_1$
(2) $18a_3+14a_2+12a_1=7a_3+3a_2+a_1=0$
(3) $15a_3+19a_2+13a_1=4a_3+8a_2+2a_1=0$
(4) $19a_3+15a_2+14a_1=8a_3+4a_2+3a_1=0$

(2) $a_1=-7a_3-3a_2=4a_3+8a_2$
(3) $4a_3+8a_2+2(4a_3+8a_2)=12a_3+24a_2=a_3+2a_2=0$
(3) $a_2=\frac{1}{2}\cdot(-a_3)=6\cdot 10a_3=60a_3=5a_3$

(4) $8a_3+4a_2+3(-7a_3-3a_2)=8a_3+4a_2+3(4a_3+8a_2)=20a_3+28a_2=\ 160a_3=6a_3=\ 5$
(4) $a_3=\frac{1}{6}\cdot 5=2\cdot 5=10$

Replacing all coefficient with the results we have found, we obtain the following (remember that we're always operating in the Z_{11} world:
$a_3 = 10$
$a_2 = 5a_3 = 50_{mod11} = 6$
$a_1 = -7a_3 - 3a_2 = 4a_3 + 8a_2 = 40 + 48 = 88_{mod11} = 0$
$a_0 = -a_3 - a_2 - a_1 = 10a_3 + 10a_2 + 10a_1 = 100 + 60 + 0 = 160_{mod11} = (11 * 14 + 6)_{mod11} = 6$

So our polynomial $A_1(x)$ becomes:

$$A_1(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0 = 10x^3 + 6x^2 + 6$$

We can easily double check that (remember that modulus operation can be applied anywhere to make easier calculations):

$$A_1(x = 1) = 10 \cdot 1^3 + 6 \cdot 1^2 + 6 = 10 + 6 + 6 = 22_{mod11} = 0$$

$$A_1(x = 2) = 10 \cdot 2^3 + 6 \cdot 2^2 + 6 = 80 + 24 + 6 = 110_{mod11} = (11 * 10)_{mod11} = 0$$

$$A_1(x = 3) = 10 \cdot 3^3 + 6 \cdot 3^2 + 6 = 270 + 54 + 6 = 330_{mod11} = (11 * 30)_{mod11} = 0$$

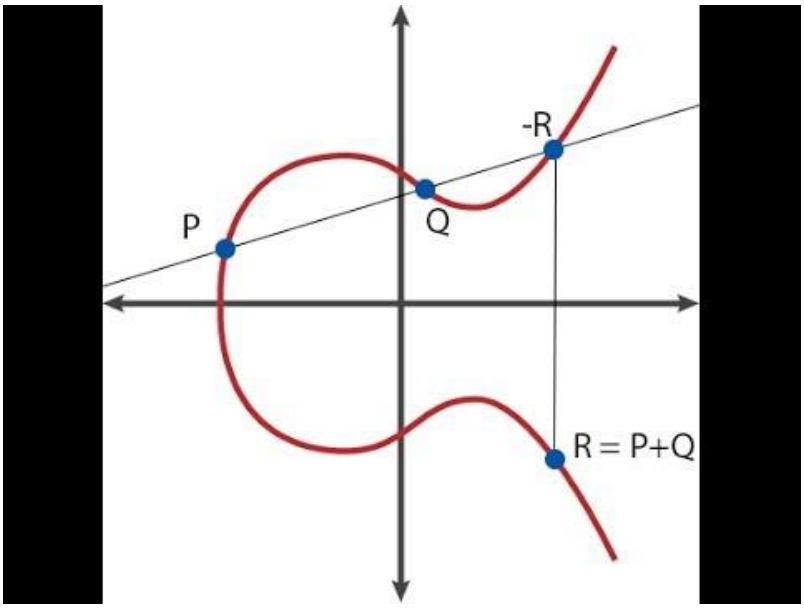
$$A_1(x = 4) = 10 \cdot 4^3 + 6 \cdot 4^2 + 6 = 10 \cdot (64_{mod11}) + 6 \cdot 16_{mod11} + 6 = 90 + 30 + 6 = 126_{mod11} = (11 * 11 + 5)_{mod11} = 5$$

3.7 Homomorphic encryption and elliptic curves

The trick to add 'privacy', is that of working in an encrypted world with the parameters that need to be passed between the prover and the verifier, or in any case to provide those parameters in the pre-staging phase, for the non-interactive proofs. Zk-Snark algorithm uses **elliptic curves**, thus the same kind of curves used by Bitcoin wallets to generate public and private keys to sign transactions.

An elliptic curve is the following:

$$y^3 = x^3 + ax + b$$



Points on the curve are on the red line, the points can be defined in the $Z_p$space of integer numbers. As usual with cryptography, as you get into the details things become really complex. Based on such curves, there's more than one cryptographic algorithm but the one used in Bitcoin is the following:

https://it.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm

Given a starting point G on the curve (the Generator), the encrypted version of the private key k is the multiplication of the two numbers, being $Q_{a=}(x_1, y_1)$ the public key.

$$(x_1, y_1) = k \cdot G$$

The operations seems to be quite simple, but we need to think about how difficult it is to reverse it. The sum operation in the mathematical group of numbers belonging to the elliptic curve, is graphically showed in the above picture for point P and Q. If you just know G and (x_1, y_1), there is no other way to find k than a brute force attack. The space of the numbers is so big that this, for non-quantic computers, would require thousands of years (see the example below to get an idea). Parameters need to be carefully chosen to have secure curves, for example:

https://en.bitcoin.it/wiki/Secp256k1

The advantage of such a complex approach, is the algorithm efficiency respect to RSA encryption algorithm (based on two prime numbers p and q in the space of integer numbers Z_{pq}).

Security (In Bits)	RSA Key Length Required (In Bits)	ECC Key Length Required (In Bits)
80	1024	160-223
112	2048	224-255
128	3072	256-383
192	7680	384-511
256	15360	512+

Some more details can be found here:

https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627

The problem of such an encryption algorithm, is that only the multiplication is preserved in the original and the encrypted world, meaning that:

$$E(x \cdot y) = E(x) \cdot E(y)$$

Since for the sum of any two values x and y this is not true, it becomes more tricky to elaborate simple zero-knowledge protocols for the verifier, because you can't simply pass a random value 'E(x)' to the prover, and the prover calculates E(f(x)), since f(x) contains addition operations that can't be later on reversed in the original world by the verifier.
Some Python code to better understand how things work has been written here (by myself):

https://github.com/ricky-andre/Bitcoin/blob/master/Bitcoin_pub_key_gen.py

… and this is just an example of the numbers used in real life applications (the private key of course is not a used one, at least not by me ☺).

```
# The proven prime
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 -1
```

```
# These two defines the elliptic curve. y^2 = x^3 + Acurve * x + Bcurve
Acurve = 0; Bcurve = 7
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424
# Generator Point
GPoint = (Gx,Gy)
# Number of points in the field
N=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
# hex version of the key below, to put it inside www.bitaddress.org
# privKey = 0xa665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
privKey = 75263518707598184987916378021939673586055614731957507592904438851787542395619
```

4 Why and How zk-SNARK Works

Despite the existence of multiple great resources on *zk-SNARK* construction, from original papers [Bit+11; Par+13] to explainers [Rei16; But16; But17; Gab17], due to the sheer number of moving parts the subject remains a black box for many. While some pieces of the puzzle are given one can not see the full picture without the missing ones.

The first time author discovered how things fit nicely together, one was astounded by the beauty of math, and the more dimensions were uncovered, the more it kept the spirit of wonderment. Hence the focus of this work is sharing the experience by shedding light onto the topic with a straightforward and clean approach based on examples and answering many whys along the way so that more individuals can appreciate the state of the art technology, its innovators and ultimately the beauty of math.

The work's contribution is a simplistic exposition with a sufficient level of complexity, necessary to understand *zk-SNARK* without any prerequisite knowledge of the subject, cryptography or advanced math. The primary goal is not only to explain how it works but why it works and how it came to be this way.

5 Factorization

The Fundamental Theorem of Algebra states that any polynomial can be factored into linear polynomials (i.e., a degree 1 polynomials representing a line), as long it is solvable. Consequently, we can represent any valid polynomial as a product of its factors:

$$(x-a_0)(x-a_1)...(x-a_n)=0$$

Also, if any of these factors is zero then the whole equation is zero, henceforth all the *a*-s are the only solutions. In fact, our example can be factored into the following polynomial:

$$x^3-3x^2+2x=(x-0)(x-1)(x-2)$$

And the solutions are (values of *x*): 0, 1, 2, you can check this easily on either form of the polynomial, but the factorized form has all the solutions (also called roots) on the surface.

Getting back to the prover's claim that he knows a polynomial of degree 3 with the roots 1 and 2, this means that his polynomial has the form:

$$(x-1)(x-2)\cdot\ldots$$

In other words $(x-1)$ and $(x-2)$ are the cofactors of the polynomial in question. Hence if the prover wants to prove that indeed his polynomial has those roots without disclosing the polynomial itself, he needs to prove

that his polynomial $p(x)$ is the multiplication of those cofactors $t(x) = (x- 1)(x- 2)$, called *target polynomial*, and some arbitrary polynomial $h(x)$ (equals to $x - 0$ in our example), i.e.:

$$p(x) = t(x) \cdot h(x)$$

In other words, there exists some polynomial $h(x)$ which makes $t(x)$ equal to $p(x)$, therefore $p(x)$ contains $t(x)$, consequently $p(x)$ has all roots of $t(x)$, the very thing to be proven. A natural way to find $h(x)$ is through the division:

$$h(x) = \frac{p(x)}{t(x)}$$

If the prover cannot find such $h(x)$ that means that $p(x)$ does not have the necessary cofactors $t(x)$, in which case the polynomials division will have a remainder. In our example if we divide $\mathbf{p(x) = x^3 - 3x^2 + 2x}$ by the $t(x) = (x - 1)(x - 2) = x^2 - 3x + 2$:

$$\begin{array}{r} x \\ \hline x^2 - 3x + 2 \big) \;\; x^3 - 3x^2 + 2x \\ -x^3 + 3x^2 - 2x \\ \hline 0 \end{array}$$

Note: the denominator is to the left, the result is to the top right, and the remainder is to the bottom (polynomial division explanation with examples is available at [Pik14]).
We have got the result $\mathbf{h(x) = x}$ without remainder.
Note: for simplicity, onwards we will use polynomial's letter variable to denote its evaluation, e.g., $p = p(r)$

Using our polynomial identity check protocol we can compare polynomials $p(x)$ and $t(x) \cdot h(x)$:

- verifier samples a random value r, calculates $\mathbf{t = t(r)}$ (i.e., evaluates) and gives r to the prover
- prover calculates $\mathbf{h(x) = p(x) / t(x)}$ and evaluates $\mathbf{p(r)}$ and $\mathbf{h(r)}$; the resulting values p, h are provided to the verifier
- verifier then checks that $p = t \cdot h$, if so those polynomials are equal, meaning that $p(x)$ has $t(x)$ as a cofactor.

To put this into practice, let us execute this protocol for our example:

- Verifier samples a random value 23, calculates $\mathbf{t = t(23) = (23 - 1)(23 - 2) = 462}$ and gives 23 to the prover
- Prover calculates $h(x) = p(x) / t(x) = x$, evaluates $p = p(23) = t(23) \cdot h(23) = 462 \cdot 23 = \mathbf{10626}$ and $h = h(23) = 23$ and provides p, h to the verifier
- Verifier then checks that $p = t \cdot h$: $10626 = 462 \cdot 23$, which is true, and therefore the statement is proven

On the contrary, if the prover uses a different $p'(x)$ which does not have the necessary cofactors, for example $p'(x) = \mathbf{2}x^3 - 3x^2 + 2x$, then:

$$h(x) = \begin{array}{r} 2x+3 \\ x^2-3x+2\,\overline{)\;2x^3-3x^2+2x} \\ \underline{-2x^3+6x^2-4x} \\ 3x^2-2x \\ \underline{-3x^2+9x-6} \\ 7x-6 \end{array}$$

Note: although the author's chief objective is simplicity, including the set of math symbols in use, it would be detrimental for further sections to omit the ubiquitous symbol prime: '. Its essential purpose is to signify some transformation or derivation of the original variable or function, e.g., if we want to multiply v by 2 and assign it to a separate variable, we could use prime: v' = 2 · v.

We will get $2x + 3$ with the remainder $7x - 6$, i.e.: $p(x) = t(x) \cdot (2x + 3) + 7x - 6$. This means that the prover will have to divide the remainder by the $t(x)$ in order to evaluate:

$$h(x) = 2x + 3 + \frac{7x-6}{t(x)}$$

Therefore because of the random selection of x by the verifier, there is a low (but still non-negligible) probability that the evaluation of the remainder $7x - 6$ will be evenly divisible by the evaluation of $t(x)$, henceforth if verifier will additionally check that p and h must be integers, such proofs will be rejected. However, the check requires the polynomial coefficients to be integers too, creating a significant limitation to the protocol. That is the reason to introduce cryptographic primitives which make such division impossible, even if the raw evaluations happen to be divisible.

Remark 3.1 *Now we can check a polynomial for specific properties without learning the polynomial itself, so this already gives us some form of zero-knowledge and succinctness. Nonetheless, there are multiple issues with this construction:*

- *Prover may not know the claimed polynomial $p(x)$ at all. He can calculate evaluation $t = t(r)$, select a random number h and set $p = t \cdot h$, which will be accepted by the verifier as valid, since equation holds.*
- *Because prover knows the random point $x = r$, he can construct any polynomial which has one shared point at r with $t(r) \cdot h(r)$.*
- *In the original statement, prover claims to know a polynomial of a particular degree, in the current protocol there is no enforcement of degree. Hence prover can cheat by using a polynomial of higher degree which also satisfies the cofactors check $p = t \cdot h$.*

We will address all of the issues in the following sections.

5.1 The Medium of a Proof

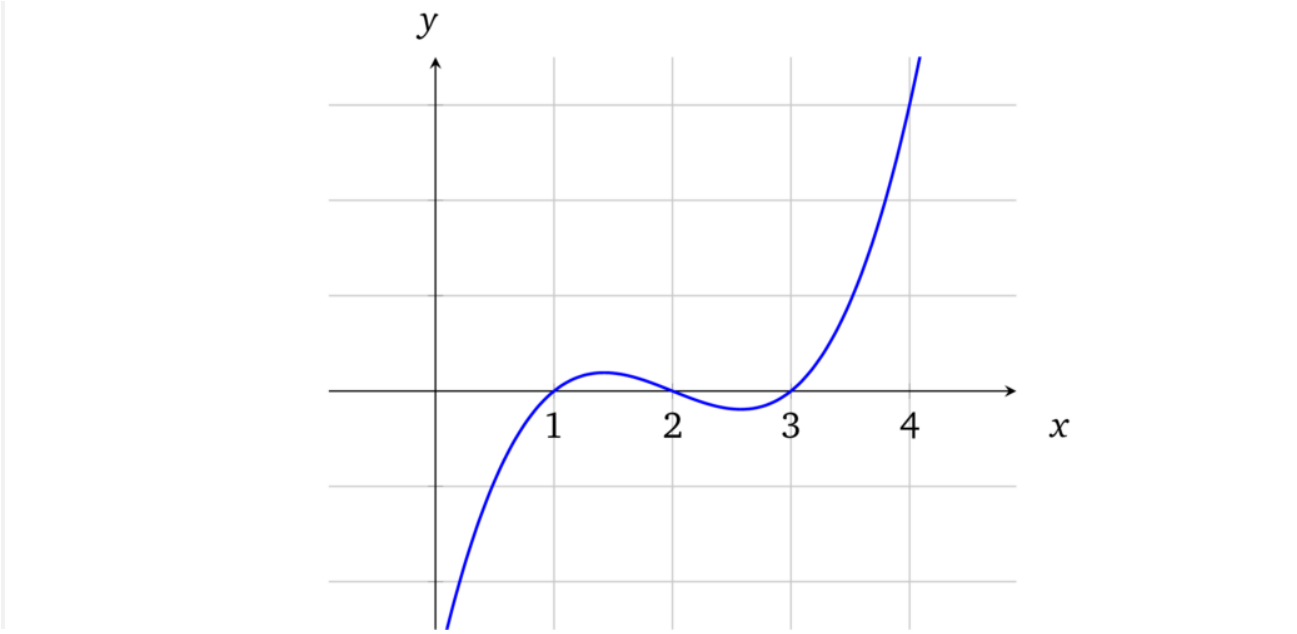
Let us start simple and try to prove something without worrying about the zero-knowledge, non-interactivity, its form, and applicability. Imagine that we have an array of bits of length 10, and we want to prove to a verifier (e.g., program) that all those bits are set to 1.

$$b = [\,?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?\,]$$

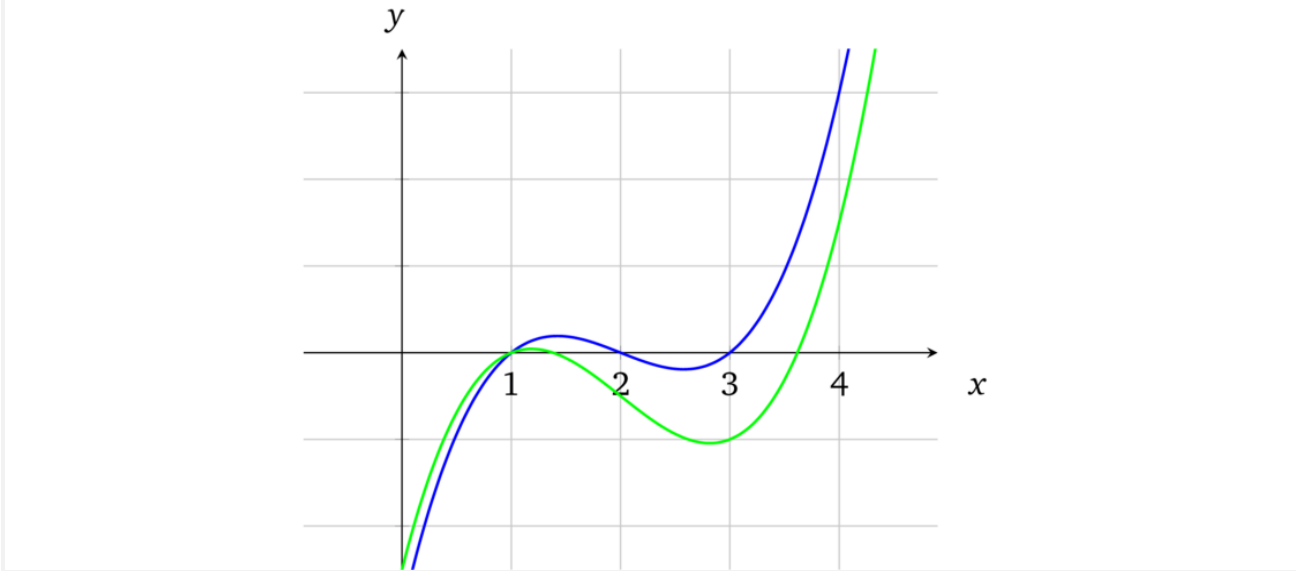
Verifier can only check (i.e., read) one element at a time. In order to verify the statement one can proceed by reading elements in some arbitrary order and checking if it is truly equal to 1 and if so the confidence in that

statement is ⅒= 10% after the first check, or statement is invalidated altogether if the bit equals to 0. A verifier must proceed to the next round until he reaches sufficient confidence. In some cases, one may trust a prover and require only 50% confidence which means that 5 checks must be executed, in other cases where 95% confidence is needed all cells must be checked. It is clear that the downside of such a proving protocol is that one must do the number of checks proportionate to the number of elements, which is non-practical if we deal with arrays of millions of elements.

Let us consider polynomials, which can be visualized as a curve on a graph, shaped by a mathematical equation:



The above curve corresponds to the polynomial: $f(x) = x^3 - 6x^2 + 11x - 6$. The degree of a polynomial is determined by its greatest exponent of x, which in this case is 3.
Polynomials have an advantageous property, namely, if we have two non-equal polynomials of degree at most d, they can intersect at no more than d points. For example, let us modify the original polynomial slightly $x^3 - 6x^2 + 10x - 5$ and visualize it in green:

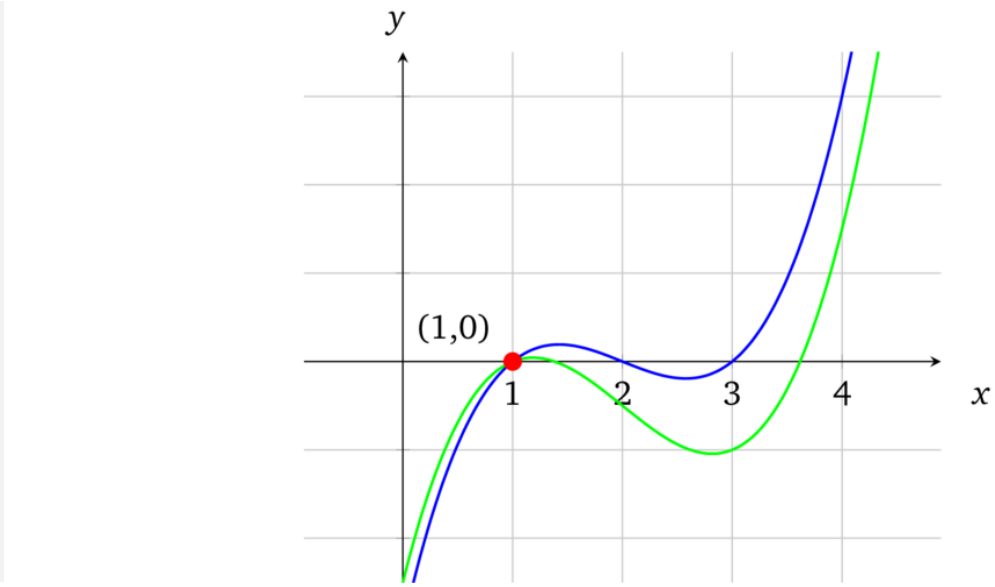


Such a tiny change produces a dramatically different result. In fact, it is impossible to find two non-equal polynomials, which share a consecutive chunk of a curve (excluding a single point chunk case).
This property flows from the method of finding shared points. If we want to find intersections of two polynomials, we need to equate them. For example, to find where a polynomial crosses an x-axis (i.e., $f(x) = 0$),

we equate $x^3 - 6x^2 + 11x - 6 = 0$, and solutions to such an equation will be those shared points: $x = 1$, $x = 2$ and $x = 3$, also you can clearly see that this is true on the previous graph, where the blue curve crosses the x-axis line.
Likewise, we can equate our original and modified version of polynomials to find their intersections.

$$x^3 - 6x^2 + 11x - 6 = x^3 - 6x^2 + \mathbf{10}x - \mathbf{5}$$
$$x - 1 = 0$$

The resulting polynomial is of degree 1 with an obvious solution $x = 1$. Hence only one intersection:



The result of any such equation for arbitrary degree d polynomials is always another polynomial of degree at most d, since there is no multiplication to produce higher degrees. Example: $5x^3 + 7x^2 - x + 2 = 3x^3 - x^2 + 2x - 5$, which simplifies to $2x^3 + 8x^2 - 3x + 7 = 0$. And the Fundamental Theorem of Algebra tells us that a degree d polynomial can have at most d solutions (more on this in following parts), and therefore at most d shared points.
Hence we can conclude that evaluation (more on polynomial evaluation: [Pik13]) of any polynomial at an arbitrary point is akin to the representation of its unique identity. Let us evaluate our example polynomials at $x = 10$.

$$x^3 - 6x^2 + 11x - 6 = 504$$
$$x^3 - 6x^2 + 10x - 5 = 495$$

In fact out of all choices of x to evaluate, only at most 3 choices will have equal evaluations in those polynomials and all others will differ. That is why if a prover claims to know some polynomial (no matter how large its degree is) that the verifier also knows, they can follow a simple protocol:

- Verifier chooses a random value for x and evaluates the polynomial locally
- Verifier gives x to the prover and asks to evaluate the polynomial in question
- Prover evaluates his polynomial at x and gives the result to the verifier
- Verifier checks if the local result is equal to the prover's result, and if so then the statement is proven with a high confidence

If we, for example, consider an integer range of x from 1 to 10^{77}, the number of points where evaluations are different is $10^{77} - d$. Henceforth the probability that x accidentally "hits" any of the d shared points is equal to (which is considered negligible):

$$\frac{d}{10^{77}}$$

Note: the new protocol requires only one round and gives overwhelming confidence (almost *100%* assuming d is sufficiently smaller than the upper bound of the range) in the statement compared to the inefficient bit check protocol. That is why polynomials are at the very core of *zk-SNARK*, although it is likely that other proof mediums exist as well.

We start with a problem of proving the knowledge of a polynomial and make our way to a generic approach. We will discover many other properties of polynomials along the way.
The discussion so far has focused on a weak notion of a proof, where parties have to trust each other because there are no measures yet to enforce the rules of the protocol. For example, the prover is not required to know a polynomial, and he can use any other means available to him to come up with a correct result. Moreover, if the amplitude of the verifier's polynomial evaluations is not large, let us say 10, the prover can guess a number, and there is a non-negligible probability that it will be accepted. We have to address such weakness of the protocol, but first what does it means to know a polynomial? A polynomial can be expressed in the form (where n is the degree of the polynomial):

$$c_n x^n + \ldots + c_1 x^1 + c_0 x^0$$

If one stated that he or she knows a polynomial of degree 1 (i.e., $c_1x^1 + c_0=0$), that means that what one really *knows* is the coefficients c_0, c_1. Moreover, coefficients can have any value, including 0.
Let us say that the prover claims to know a degree 3 polynomial, such that $x = 1$ and $x = 2$ are two of all possible solutions. One of such valid polynomials is $x^3 - 3x^2 + 2x = 0$.
For $x = 1$: $1 - 3 + 2 = 0$
For $x = 2$: $8 - 12 + 4 = 0$

Let us first look more closely at the anatomy of the solution.

5.2 Obscure Evaluation

Two first issues of remark 3.1 are possible because values are presented at raw, prover knows r and $t(r)$. It would be ideal if those values would be given as a black box, so one cannot temper with the protocol, but still able to compute operations on those obscure values. Something similar to the hash function, such that when computed it is hard to go back to the original input.

5.2.1 Homomorphic Encryption

That is exactly what homomorphic encryption is designed for. Namely, it allows to encrypt a value and be able to apply arithmetic operations on such encryption. There are multiple ways to achieve homomorphic properties of encryption, and we will briefly introduce a simple one.
The general idea is that we choose a base (there are certain properties that base number needs to have) natural number g (say 5) and to encrypt a value we exponentiate g to the power of that value. For example, if we want to encrypt the number 3:

$$5^3 = 125$$

Where 125 is the encryption of 3. If we want to multiply this encrypted number by 2, we raise it to the exponent of 2:

$$125^2 = 15625 = (5^3)^2 = 5^{2\times 3} = 5^6$$

We were able to multiply an unknown value by 2 and keep it encrypted. We can also add two encrypted values through multiplication, for example, 3 + 2:

$$5^3 \cdot 5^2 = 5^{3+2} = 5^5 = 3125$$

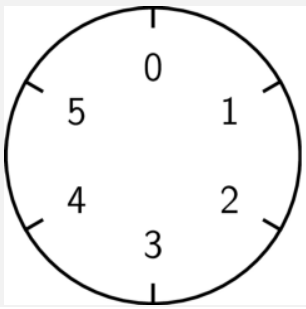
Similarly, we can subtract encrypted numbers through division, for example, 5 – 3:

$$\frac{5^5}{5^3} = 5^5 \cdot 5^{-3} = 5^{5-3} = 5^2 = 25$$

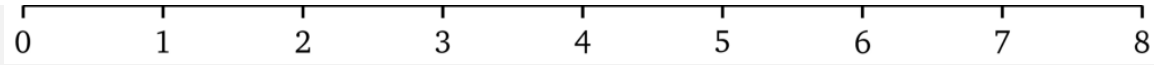
However, since the base 5 is public, it is quite easy to go back to the secret number, dividing encrypted by 5 until the result is 1. The number of steps is the secret number.

5.2.2 Modular Arithmetic

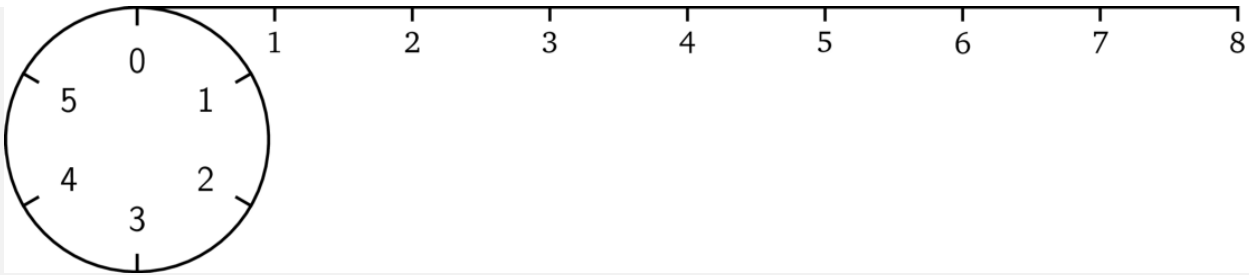
That is where the modular arithmetic comes into play. The idea of modular arithmetic is following: instead of having an infinite set of numbers we declare that we select only first *n* natural numbers, i.e., 0, 1, …, *n* – 1, to work with, and if any given integer falls out of this range, we "wrap" it around. For example, let us choose six first numbers. To illustrate this, consider a circle with six ticks of equal units; this is our range (usually referred to as finite field).



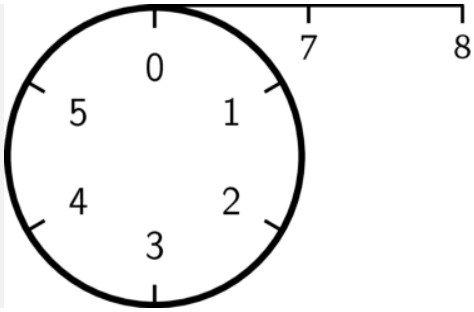
Now let us see where the number eight will land. As an analogy, we can think of it as a rope, the length of which is eight units:



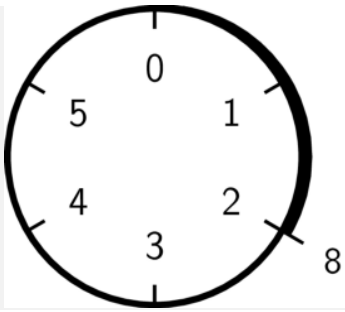
If we attach the rope to the beginning of the circle



and start wrapping the rope around it, after one rotation we still have a portion of the rope left:



Therefore if we continue the process, the rope will end right at the tick #2.



It is the result of the modulo operation. No matter how long the rope is it will always stop at one of the circle's ticks. Therefore the modulo operation will keep it in certain bounds (in this case from 0 to 5). The 15-units rope will stop at 3, i.e., 6 + 6 + 3 (two full circles with 3-units leftover). The negative numbers work the same way, and the only difference is that we wrap it in the opposite direction, for –8 the result will be 4.
Moreover, we can perform arithmetic operations, and the result will always be in the scope of *n* numbers. We will use the notation "mod *n*" for now on to denote the range of numbers. For example:

$$3 \times 5 = 3 \mod 6$$

$$5 + 2 = 1 \mod 6$$

Furthermore, the most important property is that the order of operations does not matter, e.g., we can perform all operations first and then apply modulo or apply modulo after every operation. For example $(2 \times 4 - 1) \times 3 = 3 \pmod 6$ is equivalent to:

$$2 \times 4 = 2 \pmod 6$$

$$2 - 1 = 1 \pmod 6$$

$$1 \times 3 = 3 \pmod 6$$

So why on earth is that helpful? It turns out that if we use modulo arithmetic, having a result of operation it is non-trivial to go back to the original numbers because many different combinations will have the same result:

$$5 \times 4 = 2 \pmod 6$$

$$4 \times 2 = 2 \pmod 6$$

$$2 \times 1 = 2 \pmod 6$$

$$\ldots$$

Without the modular arithmetic, the size of the result gives a clue to its solution. This piece of information is hidden otherwise, while common arithmetic properties are preserved.

5.2.3 Strong Homomorphic Encryption

If we go back to the homomorphic encryption and use modular arithmetic, for example with modulo 7, we will get:

$$5^1 = 5 \pmod 7$$
$$5^2 = 4 \pmod 7$$
$$5^3 = 6 \pmod 7$$
$$\dots$$

And different exponents will have the same result:

$$5^5 = 3 \pmod 7$$
$$5^{11} = 3 \pmod 7$$
$$5^{17} = 3 \pmod 7$$
$$\dots$$

This is where it gets *hard* to find the exponent. In fact, if modulo is sufficiently large, it becomes infeasible to do so, and a good portion of the modern-day cryptography is based on the "hardness" of this problem.
All the homomorphic properties of the scheme are preserved in the modular realm:

$$\text{encryption}: \ 5^3 = 6 \pmod 7$$
$$\text{multiplication}: \ 6^2 = (5^3)^2 = 5^6 = 1 \pmod 7$$
$$\text{addition}: \ 5^3 \cdot 5^2 = 5^5 = 3 \pmod 7$$

Note: modular division is a bit more complicated and out of the scope.
Let us explicitly state the encryption function:

$$E(v) = g^v \pmod n$$

where *v* is the value we want to encrypt.
Remark 3.2 *There are limitations to this homomorphic encryption scheme while we can multiply an encrypted value by an unencrypted value, we cannot multiply (and divide) two encrypted values, as well as we cannot exponentiate an encrypted value. While unfortunate from the first impression, these properties will turn out to be the cornerstone of zk-SNARK. The limitations are addressed in section "Multiplication of Encrypted Values".*

5.3 Encrypted Polynomial

Armed with such tools, we can now evaluate a polynomial with an encrypted random value of *x* and modify the *zero-knowledge* protocol accordingly.
Let us see how we can evaluate a polynomial $p(x) = x^3 - 3x^2 + 2x$. As we have established previously to know a polynomial is to know its coefficients, in this case those are: 1, –3, 2. Because homomorphic encryption does not allows to exponentiate an encrypted value, we've must been given encrypted values of powers of *x* from 1 to 3: $E(x)$, $E(x^2)$, $E(x^3)$, so that we can evaluate the encrypted polynomial as follows:

$$E\left(x^3\right)^1 \cdot E\left(x^2\right)^{-3} \cdot E\left(x\right)^2 =$$
$$\left(g^{x^3}\right)^1 \cdot \left(g^{x^2}\right)^{-3} \cdot \left(g^{x}\right)^2 =$$
$$g^{1x^3} \cdot g^{-3x^2} \cdot g^{2x} =$$
$$g^{x^3-3x^2+2x}$$

As the result of such operations, we have an encrypted evaluation of our polynomial at some unknown to us *x*. This is quite a powerful mechanism, and because of the homomorphic property, the encrypted evaluations of the same polynomials are always the same in encrypted space.
We can now update the previous version of the protocol, for a polynomial of degree *d*:

- Verifier
  - samples a random value s, i.e., secret
  - calculates encryptions of s for all powers i in $0, 1, \ldots, d$, i.e.: $E(s^i) = g^{s^i}$
  - evaluates unencrypted *target polynomial* with s: $t(s)$
  - encrypted powers of s are provided to the prover: $E(s^0), E(s^1), \ldots, E(s^d)$
- Prover
  - calculates polynomial $h(x) = \frac{p(x)}{t(x)}$
  - using encrypted powers $g^{s^0}, g^{s^1}, \ldots, g^{s^d}$ and coefficients $c_0, c_1, \ldots, c_n$ evaluates

    $E(p(s)) = g^{p(s)} = \left(g^{s^d}\right)^{c_d} \cdots \left(g^{s^1}\right)^{c_1} \cdot \left(g^{s^0}\right)^{c_0}$ and similarly $E(h(s)) = g^{h(s)}$
  - the resulting g^p and g^h are provided to the verifier
- Verifier
  - The last step for the verifier is to checks that $p = t(s) \cdot h$:

    $g^p = \left(g^h\right)^{t(s)} \quad \Rightarrow \quad g^p = g^{t(s) \cdot h}$

Note: because the prover does not know anything about s, it makes it hard to come up with non-legitimate but still matching evaluations.

While in such protocol the prover's agility is limited he still can use any other means to forge a proof without actually using the provided encryptions of powers of *s*, for example, if the prover claims to have a satisfactory polynomial using only 2 powers s^3 and s^1, that is not possible to verify in the current protocol.

5.4 Restricting a Polynomial

The knowledge of a polynomial is the knowledge of its coefficients $c_0, c_1, \ldots, c_i$ and the way we "assign" those coefficients in the protocol is through exponentiation of the corresponding encrypted powers of the secret value *s*. We do already restrict a prover in the selection of encrypted powers of *s*, but such restriction is not enforced, e.g., one could use any possible means to find some arbitrary values Zp and *Zh* which satisfy equation:

$$z_p = (z_h)^{t(s)}$$

… and provide them to the verifier instead of g^p and g^h. That is why verifier needs the proof that only encryptions of powers of *s* were used and nothing else.
Let us consider an elementary example of a degree 1 polynomial with one variable and one coefficient $f(x) = c \cdot x$ and correspondingly the encryption of the *s* is provided $E(s) = g^s$. What we are looking for is to make sure that only encryption of *s*, i.e., g^s, was homomorphically "multiplied" by some arbitrary coefficient *c* and nothing else. So the result must always be of the form (for some arbitrary *c*):

$$(g^s)^c$$

A way to do this is to require to perform the same operation on another *shifted* encrypted value alongside with the original one, acting as an arithmetic analog of "checksum", ensuring that the result is exponentiation of the original value.
This is achieved through the Knowledge-of-Exponent Assumption (or KEA), introduced in [Dam91], more precisely (note the difference between *a* and α (alpha)):

a) Alice has a value *a*, that she wants Bob to exponentiate to any power (where *a* is a generator of a finite field group used), the single requirement is that only this *a* can be exponentiated and nothing else, to ensure this she:

- chooses a random α
- calculates $a' = a^{\alpha} \pmod{n}$
- provides the tuple (a, a') to Bob and asks to perform same arbitrary exponentiation of each value and reply with the resulting tuple (b, b') where the exponent "α-shift" remains the same, i.e., $b^{\alpha} = b' \pmod{n}$

b) because Bob cannot extract α from the tuple (a, a') other then through a brute-force which is infeasible, it is conjectured that the only way Bob can produce a valid response is through the procedure:

- chose some value c
- calculate $b = (a)^c \pmod{n}$ and $b' = (a')^c \pmod{n}$
- reply with (b, b')

c) having the response and α, Alice checks the equality:

$$(b)^{\alpha} = b'$$
$$(a^c)^{\alpha} = (a')^c$$
$$a^{c\cdot\alpha} = (a^{\alpha})^c$$

Conclusions:

- Bob has applied the same exponent (i.e., *c*) to both values of the tuple
- Bob could only use the original Alice's tuple to maintain the α relationship
- Bob *knows* the applied exponent *c*, because the only way to produce valid (b,b') is to use the same exponent
- Alice has not learned *c* for the same reason Bob cannot learn α *.

* Although the *c* is *encrypted* its range of possible values might not be sufficient to preserve *zero-knowledge* property which will be addressed in the section "Zero Knowledge".

Ultimately such protocol provides a proof to Alice that Bob indeed exponentiated *a* by some value known to him, and he could not do any other operation, e.g., multiplication, addition, since this would erase the α-shift relationship. In the homomorphic encryption context, exponentiation is the multiplication of the encrypted value. We can apply the same construction in the case with the simple one-coefficient polynomial $f(x) = c \cdot x$:

- Verifier chooses random s,α and provides evaluation for $x = s$ for power 1 and its "shift":
$$(g^s, g^{\alpha\cdot s})$$
- Prover applies the coefficient *c*:
$$((g^s)^c, (g^{\alpha\cdot s})^c) = (g^{c\cdot s}, g^{\alpha\cdot c\cdot s})$$
- Verifier checks:
$$(g^{c\cdot s})^{\alpha} = g^{\alpha\cdot c\cdot s}$$

Such construction *restricts* the prover to use only the encrypted *s* provided, therefore prover could have assigned coefficient *c* only to the polynomial provided by the verifier. We can now scale such one-term polynomial (monomial) approach to a multi-term polynomial because the coefficient assignment of each term is calculated separately and then homomorphically "added" together (this approach was introduced by Jens Groth in [Gro10]). So if the prover is given encrypted exponentiations of *s* alongside with their *shifted* values he can evaluate original and shifted polynomial, where the same check must hold. In particular, for a degree *d* polynomial:

- Verifier provides encrypted powers $g^{s^0}, g^{s^1}, \ldots, g^{s^d}$ and their shifts $g^{\alpha s^0}, g^{\alpha s^1}, \ldots, g^{\alpha s^d}$
- Prover:
  - evaluates encrypted polynomial with provided powers of s:
    $$g^{p(s)} = \left(g^{s^0}\right)^{c_0} \cdot \left(g^{s^1}\right)^{c_1} \cdot \ldots \cdot \left(g^{s^d}\right)^{c_d} = g^{c_0 s^0 + c_1 s^1 + \ldots + c_d s^d}$$
  - evaluates encrypted "shifted" polynomial with the corresponding α-shifts of the powers of s:
    $$g^{\alpha p(s)} = \left(g^{\alpha s^0}\right)^{c_0} \cdot \left(g^{\alpha s^1}\right)^{c_1} \cdot \ldots \cdot \left(g^{\alpha s^d}\right)^{c_d} = g^{c_0 \alpha s^0 + c_1 \alpha s^1 + \ldots + c_d \alpha s^d} = g^{\alpha(c_0 s^0 + c_1 s^1 + \ldots + c_d s^d)}$$
  - provides the result as $g^p, g^{p'}$ to the verifier
- Verifier checks: $(g^p)^{\alpha} = g^{p'}$

For our previous example polynomial $p(x) = x^3 - 3x^2 + 2x$ this would be:

- Verifier provides $E(s^3), E(s^2), E(s)$ and their shifts $E(\alpha s^3), E(\alpha s^2), E(\alpha s)$
- Prover evaluates:

$$g^p = g^{p(s)} = \left(g^{s^3}\right)^1 \cdot \left(g^{s^2}\right)^{-3} \cdot \left(g^s\right)^2 = g^{s^3} \cdot g^{-3s^2} \cdot g^{2s} = g^{s^3-3s^2+2s}$$

$$g^{p'} = g^{\alpha p(s)} = \left(g^{\alpha s^3}\right)^1 \cdot \left(g^{\alpha s^2}\right)^{-3} \cdot \left(g^{\alpha s}\right)^2 = g^{\alpha s^3} \cdot g^{-3\alpha s^2} \cdot g^{2\alpha s} = g^{\alpha(s^3-3s^2+2s)}$$

- Verifier checks $(g^p)^\alpha = g^{p'}$:

$$\left(g^{s^3-3s^2+2s}\right)^\alpha = g^{\alpha(s^3-3s^2+2s)}$$

$$g^{\alpha(s^3-3s^2+2s)} = g^{\alpha(s^3-3s^2+2s)}$$

Now we can be sure that the prover did not use anything else other than the provided by verifier polynomial, since there is no other way to preserve the α-shift. Also if a verifier would want to ensure exclusion of some power(s) of s in a prover's polynomial, e.g., j, he will not provide the encryption and its shift:

$$g^{s^j}, g^{\alpha s^j}$$

Compared to what we have started with, we now have a robust protocol. However there is still a significant drawback to the *zero-knowledge* property, regardless of encryption: while theoretically polynomial coefficients c_i can have a vast range of values, in reality, it might be quite limited (6 in the previous example), which means that the verifier could brute-force limited range of coefficients combinations until the result is equal to the prover's answer. For instance if we consider the range of 100 values for each coefficient, the degree 2 polynomial would total to 1 million of distinct combinations, which considering brute-force would require less than a million iterations. Moreover, the secure protocol should be secure even in cases where there is only one coefficient, and its value is 1.

6 Zero-Knowledge

Because verifier can extract knowledge about the unknown polynomial $p(x)$ only from the data sent by the prover, let us consider those provided values (the proof):

$$g^p, g^{p'}, g^h$$

They participate in the following checks:

$$g^p = \left(g^h\right)^{t(s)} \qquad \text{(polynomial } p(x) \text{ has roots of } t(x)\text{)}$$

$$(g^p)^\alpha = g^{p'} \qquad \text{(polynomial of a correct form is used)}$$

The question is how do we alter the proof such that the checks still hold, but no knowledge can be extracted? One answer can be derived from the previous section: we can "shift" those values by some random number δ (delta), e.g.,$(g^p)^\delta$. Now, in order to extract the knowledge, one first needs to find δ which is considered infeasible. Moreover, such randomization is statistically indistinguishable from random.

To maintain relationships let us examine the verifier's checks. One of the prover's values is on each side of the equations. Therefore if we "shift" each of them with the same δ the equations must remain balanced. Concretely, prover samples a random δ and exponentiates his proof values with it

$$\left(g^{p(s)}\right)^{\delta}, \left(g^{h(s)}\right)^{\delta}, \left(g^{\alpha p(s)}\right)^{\delta}$$

and provides to the verifier for verification:

$$(g^{p})^{\delta} = \left((g^{h})^{\delta}\right)^{t(s)}$$
$$\left((g^{p})^{\delta}\right)^{\alpha} = \left(g^{p'}\right)^{\delta}$$

After consolidation we can observe that the check still holds:

$$g^{\delta \cdot p} = g^{\delta \cdot t(s) h}$$
$$g^{\delta \cdot \alpha p} = g^{\delta \cdot p'}$$

Note: how easily the zero-knowledge is woven into the construction, this is often referred to as "free" zero-knowledge.

7 Non-Interactivity

Till this point, we had an *interactive* zero-knowledge scheme. Why is that the case? Because the proof is only valid for the original verifier, nobody else (other verifiers) can trust the same proof since:

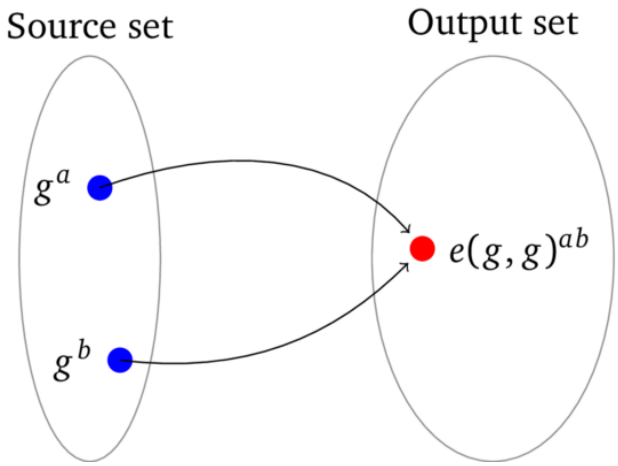
- the verifier could collude with the prover and disclose those secret parameters s,α which allows to fake the proof, as mentioned in remark 3.1
- the verifier can generate fake proofs himself for the same reason
- verifier have to store α and $t(s)$ until all relevant proofs are verified, which allows an extra attack surface with possible leakage of secret parameters

Therefore a separate interaction with every verifier is required in order for a statement (knowledge of polynomial in this case) to be proven. While interactive proof system has its use cases, for example when a prover wants to convince only a dedicated verifier (called designated verifier, more information in [JSI96]) such that the proof cannot be re-used to prove same statement to others, it is quite inefficient when one needs to convince many parties simultaneously (e.g., in distributed systems such as blockchain) or permanently. Prover would be required to stay online at all times and perform the same computation for every verifier.

Hence, we need the secret parameters to be reusable, public, trustworthy and infeasible to abuse.
Let us first consider how would we secure the secrets $(t(s),\alpha)$ after they are produced. We can encrypt them the same way verifier encrypts powers of s before sending to the prover. However as mentioned in the remark 3.2, the homomorphic encryption we use does not support the multiplication of two encrypted values, which is necessary for both verification checks to multiply encryptions of $t(s)$ and h as well as p and α. This is where cryptographic pairings fit in.

7.1 Multiplication of Encrypted Values

Cryptographic pairings (bilinear map) is a mathematical construction, denoted as a function $e(g^*,g^*)$, which given two encrypted inputs (e.g., g^a, g^b) from one set of numbers allows to map them deterministically to their multiplied representation in a different output set of numbers, i.e., $e(g^a, g^b) = e(g, g)^{ab}$:



Because the *source* and *output* number sets (usually referred to as a group*)* are different the result of the pairing is not usable as an input for another pairing operation. We can look at the output set (also called "target set") as being from a "different universe." Therefore we cannot multiply the result by another encrypted value and suggested by the name itself we can only multiply two encrypted values at a time.
In some sense, it resembles a hash function, which maps all possible input values to an element in the set of possible output values and it is not trivially reversible.
Note: from first glance, such limitation must only impede a dependent functionality, ironically in the zk-SNARK case it is a paramount property on which security of the scheme holds, see remark 3.3.
A rudimentary (and technically incorrect) mathematical analogy for pairing function $e(g^*,g^*)$ would be to state that there is a way to "swap" each input's base and exponent, such that base g is modified in the process of transformation into exponent, e.g., $g^a \rightarrow a^{\mathbf{g}}$. Both "swapped" inputs are then multiplied together, such that raw a and b values get multiplied under the same exponent, e.g.:

$$e(g^a, g^b) = a^{\mathbf{g}} \cdot b^{\mathbf{g}} = (ab)^{\mathbf{g}}$$

Therefore because the base gets altered during the "swap" using the result $(ab)^{\mathbf{g}}$ in another pairing (e.g., $e((ab)^{\mathbf{g}}, g^d)$) would not produce desired encrypted multiplication abd. The core properties of pairings can be expressed in the equations:

$$e(g^a, g^b) = e(g^b, g^a) = e(g^{ab}, g^1) = e(g^1, g^{ab}) = e(g^1, g^a)^b = e(g^1, g^1)^{ab} = \dots$$

Technically the result of a pairing is an encrypted product of raw values under a different generator $\mathbf{g}$ of the target set, i.e., $e(g^a, g^b) = \mathbf{g}^{ab}$. Therefore it has properties of the homomorphic encryption, e.g., we can add the encrypted products of multiple pairings together:

$$e(g^a, g^b) \cdot e(g^c, g^d) = \mathbf{g}^{ab} \cdot \mathbf{g}^{cd} = \mathbf{g}^{ab+cd} = e(g, g)^{ab+cd}$$

Note: cryptographic pairing is leveraging elliptic curves to achieve these properties, therefore from now on notation g^n will represent a generator point on a curve added to itself n times instead of a multiplicative group generator which we have used in previous sections.
The survey [DBS04] provides a starting point for exploration of the cryptographic pairings.

7.2 Trusted Party Setup

Having cryptographic pairings, we are now ready to set up secure public and reusable parameters. Let us assume that we trust a single honest party to generate secrets s and α. As soon as α and all necessary powers of s with corresponding α-shifts are encrypted, the raw values must be deleted (for i in 0, 1, …, d):

$$g^{\alpha}, g^{s^i}, g^{\alpha s^i}$$

These parameters are usually referred to as ***common reference string*** or **CRS**. After CRS is generated any prover and any verifier can use it in order to conduct non-interactive zero-knowledge proof protocol. While non-crucial, the optimized version of CRS will include encrypted evaluation of the *target polynomial* $g^{t(s)}$. Moreover CRS is divided into two groups (for i in 0, 1, …, d):

- Proving key (also called *evaluation key*): $(g^{s^i}, g^{\alpha s^i})$
- Verification key: $(g^{t(s)}, g^{\alpha})$

Being able to multiply encrypted values the verifier can check the polynomials in the last step of the protocol:

- Having verification key verifier processes received encrypted polynomial evaluations $g^{p}, g^{h}, g^{p'}$ from the prover:
  - checks that $p = t \cdot h$ in encrypted space:

    $e(g^{p}, g^{1}) = e(g^{t}, g^{h})$ which is equivalent to $e(g,g)^{p} = e(g,g)^{t \cdot h}$
  - checks polynomial restriction:

    $e(g^{p}, g^{\alpha}) = e(g^{p'}, g)$

7.3 Trusting One out of Many

While the trusted setup is efficient, it is not effective since multiple users of CRS will have to trust that one deleted α and s, since currently there is no way to prove that (proof of ignorance is an area of active research [DK18]). Hence it is necessary to minimize or eliminate that trust. Otherwise, a dishonest party would be able to produce fake proofs without being detected.

One way to achieve that is by generating a *composite* CRS by multiple parties employing mathematical tools introduced in previous sections, such that neither of those parties knows the secret. Here is an approach, let us consider three participants Alice, Bob and Carol with corresponding indices A, B and C, for i in 1, 2, …, d:

- Alice samples her random s_A and α_A and publishes her CRS:
  $\left(g^{s_A^i}, g^{\alpha_A}, g^{\alpha_A s_A^i}\right)$
- Bob samples his s_B and α_B and augments Alice's encrypted CRS through homomorphic multiplication:
  $\left(\left(g^{s_A^i}\right)^{s_B^i}, (g^{\alpha_A})^{\alpha_B}, \left(g^{\alpha_A s_A^i}\right)^{\alpha_B s_B^i}\right) = \left(g^{(s_A s_B)^i}, g^{\alpha_A \alpha_B}, g^{\alpha_A \alpha_B (s_A s_B)^i}\right)$
  and publishes the resulting two-party Alice-Bob CRS:
  $\left(g^{s_{AB}^i}, g^{\alpha_{AB}}, g^{\alpha_{AB} s_{AB}^i}\right)$
- So does Carol with her s_C and α_C:
  $\left(\left(g^{s_{AB}^i}\right)^{s_C^i}, \left(g^{\alpha_{AB}}\right)^{\alpha_C}, \left(g^{\alpha_{AB} s_{AB}^i}\right)^{\alpha_C s_C^i}\right) = \left(g^{(s_A s_B s_C)^i}, g^{\alpha_A \alpha_B \alpha_C}, g^{\alpha_A \alpha_B \alpha_C (s_A s_B s_C)^i}\right)$
  and publishes Alice-Bob-Carol CRS:
  $\left(g^{s_{ABC}^i}, g^{\alpha_{ABC}}, g^{\alpha_{ABC} s_{ABC}^i}\right)$

As the result of such protocol, we have composite s^i and α, where:

$$s^i = s_A^i s_B^i s_C^i, \quad \alpha = \alpha_A \alpha_B \alpha_C$$

and no participant learns secret parameters of other participants unless they are colluding. In fact, in order to learn s and α, one must collude with every other participant. Therefore even if one out of all is honest, it will be infeasible to produce fake proofs.
Note: this process can be repeated for as many participants as necessary.

The question one might have is how to verify that participant have been consistent with every value of CRS, because an adversary can sample multiple different $s_1, s_2, \ldots$ and $\alpha_1, \alpha_2, \ldots$, and use those randomly for different powers of s (or provide random numbers as an augmented common reference string), rendering CRS invalid and unusable. Luckily, because we can multiply encrypted values using pairings, we are able to perform consistency check, starting with the first parameter and ensuring that every next is derived from it. Every published CRS by participants can be checked as follows:

- We take power 1 of s as canonical value and check every other power for consistency with it:
  $e\left(g^{s^i}, g\right) = e\left(g^{s^1}, g^{s^{i-1}}\right)\Big|_{i \in \{2,\ldots,d\}}$
  for example:
  - Power 2: $e\left(g^{s^2}, g\right) = e\left(g^{s^1}, g^{s^1}\right) \Rightarrow e(g,g)^{s^2} = e(g,g)^{s^{1+1}}$
  - Power 3: $e\left(g^{s^3}, g\right) = e\left(g^{s^1}, g^{s^2}\right) \Rightarrow e(g,g)^{s^3} = e(g,g)^{s^{1+2}}$, etc.

- We now check if the α-shift of values in the previous step is correct:

$$e\left(g^{s^i}, g^{\alpha}\right) = e\left(g^{\alpha s^i}, g\right)\Big|_{i\in[d]}$$

for example:

– Power 3: $e\left(g^{s^3}, g^{\alpha}\right) = e\left(g^{\alpha s^3}, g\right) \Rightarrow e(g,g)^{s^3\cdot\alpha} = e(g,g)^{\alpha s^3}$, etc.

where $i \in \{2, \ldots, d\}$ is a shortened form of "i is in 2, 3, …, d" and $[d]$ is a shortened form of range 1, 2, …, d, which is the more convenient notation for the next sections.
Notice that while we verify that every participant is consistent with their secret parameters, the requirement to use previously published CRS is not enforced for every next party (Bob and Carol in our example). Hence if an adversary is the last in the chain he can ignore the previous CRS and construct valid parameters from scratch, as if he was the first in the chain, therefore being the only one who knows secret s and α.
We can address this by additionally requiring every participant except the first one to encrypt and publish his secret parameters, for example, Bob also publishes:

$$\left(g^{s_B^i}, g^{\alpha_B}, g^{\alpha_B s_B^i}\right)\Big|_{i\in[d]}$$

This allows to validate that Bob's CRS is a proper multiple of Alice's parameters, for i in 1, 2,…, d:

- $e\left(g^{s_{\mathrm{AB}}^i}, g\right) = e\left(g^{s_A^i}, g^{s_B^i}\right)$
- $e\left(g^{\alpha_{\mathrm{AB}}}, g\right) = e\left(g^{\alpha_A}, g^{\alpha_B}\right)$
- $e\left(g^{\alpha_{\mathrm{AB}} s_{\mathrm{AB}}^i}, g\right) = e\left(g^{\alpha_A s_A^i}, g^{\alpha_B s_B^i}\right)$

Similarly Carol will have to prove that her CRS is a proper multiple of Alice-Bob's CRS.
This is a robust CRS setup scheme which does not rely entirely on any single party. In fact, it is sufficient if only one party is honest and deletes and never shares its secret parameters, even if all other parties have colluded. So the more there are unrelated participants in CRS setup (sometimes called ceremony [Wil16]) the faintest the possibility of fake proofs, the probability becomes negligible if competing parties are participating. The scheme allows involving other untrusted parties who are in doubt about the legibility of the setup because verification step ensures they are not sabotaging (which also includes usage of weak α and s) the final common reference string.

8 Succinct Non-Interactive Argument of Knowledge of Polynomial

We are now ready to consolidate the evolved *zk-SNARKOP* protocol. Being formal, for brevity, we will be using curly brackets to denote a set of elements populated by the subscript next to it, for example:

$$\{s^i\}_{i\in[d]}$$

denotes a set $s^1, s^2, \ldots, s^d$. Having agreed upon *target polynomial* $t(x)$ and degree d of the prover's polynomial:

- Setup
  - sample random values s, α
  - calculate encryptions g^{α} and $\left\{g^{s^i}\right\}_{i\in[d]}, \left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}$
  - proving key: $\left(\left\{g^{s^i}\right\}_{i\in[d]}, \left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}\right)$
  - verification key: $\left(g^{\alpha}, g^{t(s)}\right)$
- Proving
  - assign coefficients $\{c_i\}_{i\in\{0,\ldots,d\}}$ (i.e., knowledge), $p(x) = c_d x^d + \cdots + c_1 x^1 + c_0 x^0$
  - calculate polynomial $h(x) = \frac{p(x)}{t(x)}$
  - evaluate encrypted polynomials $g^{p(s)}$ and $g^{h(s)}$ using $\left\{g^{s^i}\right\}_{i\in[d]}$
  - evaluate encrypted shifted polynomial $g^{\alpha p(s)}$ using $\left\{g^{\alpha s^i}\right\}_{i\in\{0,\ldots,d\}}$
  - sample random δ
  - set the randomized proof $\pi = \left(g^{\delta p(s)}, g^{\delta h(s)}, g^{\delta \alpha p(s)}\right)$
- Verification
  - parse proof π as $\left(g^{p}, g^{h}, g^{p'}\right)$
  - check polynomial restriction $e\left(g^{p'}, g\right) = e\left(g^{p}, g^{\alpha}\right)$
  - check polynomial cofactors $e\left(g^{p}, g\right) = e\left(g^{t(s)}, g^{h}\right)$

Remark 3.3 *If it would be possible to reuse result of pairing for another multiplication such protocol would be completely insecure because the prover can assign:*

$$g^{p'} = e\left(g^{p}, g^{\alpha}\right)$$

which would then pass the "polynomial restriction" check:

$$e\left(e\left(g^{p}, g^{\alpha}\right), g\right) = e\left(g^{p}, g^{\alpha}\right)$$

9 Conclusions

We came to the zero-knowledge succinct non-interactive arguments of knowledge protocol for the knowledge of a polynomial problem, which is a niche use-case. While one can claim that a prover can easily construct such polynomial $p(x)$ just by multiplying $t(x)$ by another bounded polynomial to make it pass the test, the construction is still useful.
Verifier knows that the prover has a valid polynomial but not which particular one. We could add additional proofs of other properties of the polynomial such as: divides by multiple polynomials, is a square of a polynomial. There could be a service which accepts, stores and rewards all the attested polynomials, or there is a need in an encrypted evaluation of unknown polynomials of a necessary form. However, having universal scheme would allow for a myriad of applications.

We have paved our way with a simple yet sufficient example involving most of the *zk-SNARK* machinery, and it is now possible to advance the scheme to execute zero-knowledge programs.

10 Computation

Let us consider a simple program in pseudocode:

Algorithm 1: Operation depends on an input

```
function calc(w, a, b)
  if w then
    return a × b
  else
    return a + b
  end if
end function
```

From a high-level view, it is quite unrelated to polynomials, which we have the protocol for. Therefore we need to find a way to convert a program into the polynomial form. The first step then is to translate the program into the language of math, which is relatively easy, the same statement can be expressed as following (assuming w is either 0 or 1):

$$f(w,a,b) = w(a \times b) + (1-w)(a+b)$$

Executing calc(1, 4, 2) and evaluating $f(1, 4, 2)$ will yield the same result: 8. Conversely calc(0, 4, 2) and $f(0,4,2)$ would both be resolved to 6. We can express any kind of finite program in such a way.
What we need to prove then (in this example), is that for the input $(1, 4, 2)$ of expression $f(w,a,b)$ the output is 8, in other words, we check the equality:

$$w(a \times b) + (1-w)(a+b) = 8$$

10.1 Single Operation

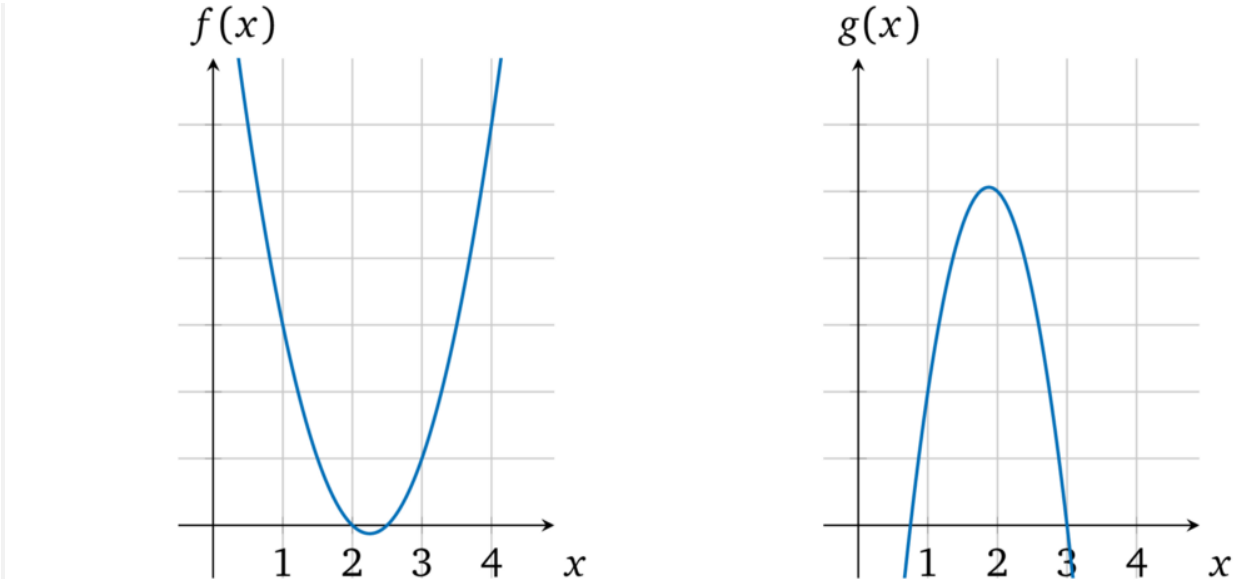
We now have a general computation expressed in a mathematical language, but we still need to translate it into the realm of polynomials. Let us have a closer look at what computation is in a nutshell. Any computation at it is core consists of elemental operations of the form:

left operand **operator** right operand = output

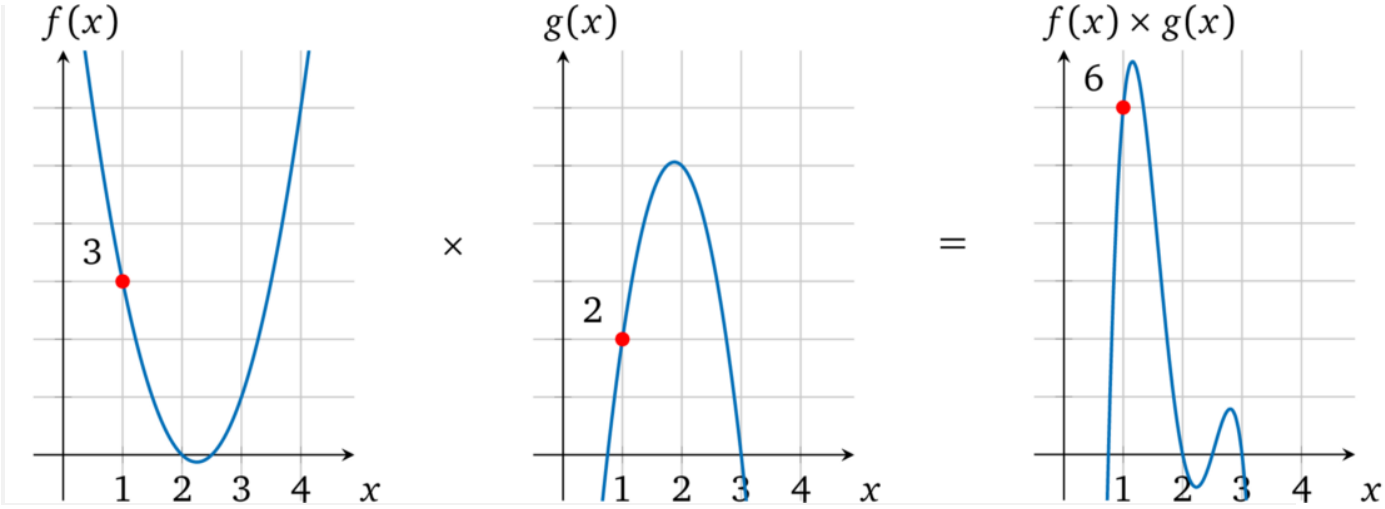
Two operands (i.e., values) are being operated upon by an operator (e.g., +,–,×,÷). For example for operands 2 and 3 and operator "multiplication" these will resolve to 2 × 3 = 6. Because any complex computation (or a program) is just a series of operations, firstly we need to find out how single such operation can be represented by a polynomial.

10.2 Arithmetic Properties of Polynomials

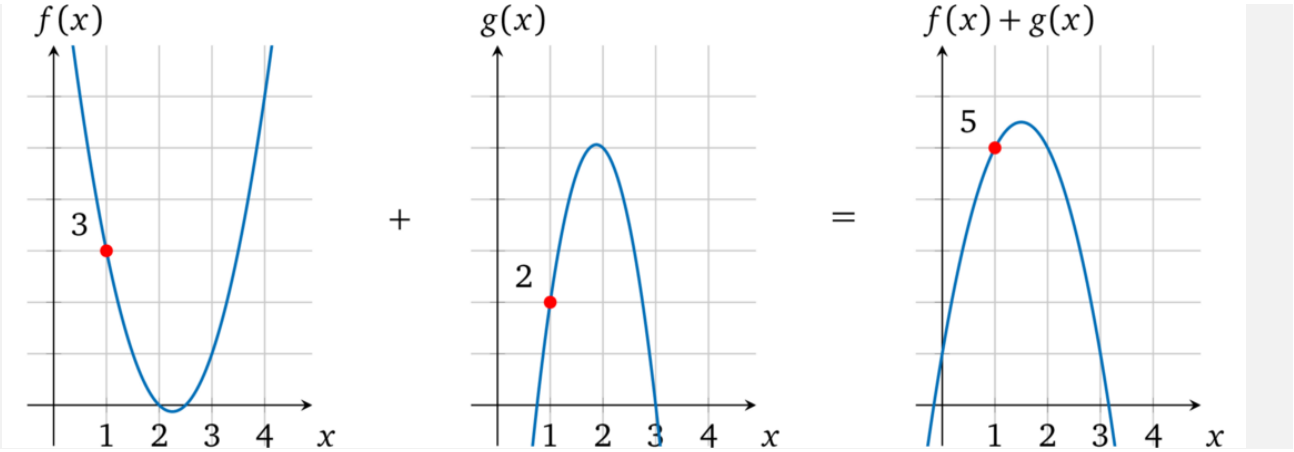
Let us see how polynomials are related to arithmetic operations. If you take two polynomials $f(x)$ and $g(x)$ and try, for example, to multiply them $h(x) = f(x) \times g(x)$, the result of evaluation of $h(x)$ at any $x = r$ will be the multiplication of results of evaluations of $f(r)$ and $g(r)$. Let us consider two following polynomials: $f(x) = 2x^2 - 9x + 10$ and $g(x) = -4x^2 + 15x - 9$. Visualized in the form of graph:



For $x = 1$ these will evaluate to: $f(1) = 2 - 9 + 10 = 3$, $g(1) = -4 + 15 - 9 = 2$. Let us multiply the polynomials: $h(x) = f(x) \times g(x) = -8x^4 + 66x^3 - 193x^2 + 231x - 90$. Visually multiplication can be seen as:



If we examine evaluations at $x = 1$ on the resulting polynomial $f(x) \times g(x)$ we will get: $h(1) = -8 + 66 - 193 + 231 - 90 = 6$, hence the values at $x = 1$ of $f(x)$ and $g(x)$ has multiplied, and respectively at every other x. Likewise if we add $f(x)$ and $g(x)$ we will get $-2x^2 + 6x + 1$ which evaluates to 5 at $x = 1$.



Note: evaluations at other x-s were also added together, e.g., examine x = 2, x = 3.

If we can represent operand values as polynomials (and we indeed can as outlined) then through the arithmetic properties, we will be able to get the result of an operation imposed by an operand.

10.3 Enforcing Operation

If a prover claims to have the result of multiplication of two numbers how does verifier checks that? To prove the correctness of a single operation, we must enforce the correctness of the output (result) for the operands provided. If we look again at the form of operation:

$$\text{left operand} \quad \textbf{operator} \quad \text{right operand} \quad = \quad \text{output}$$

The same can be represented as an *operation polynomial*:

$$l(x) \textbf{ operator } r(x) = o(x)$$

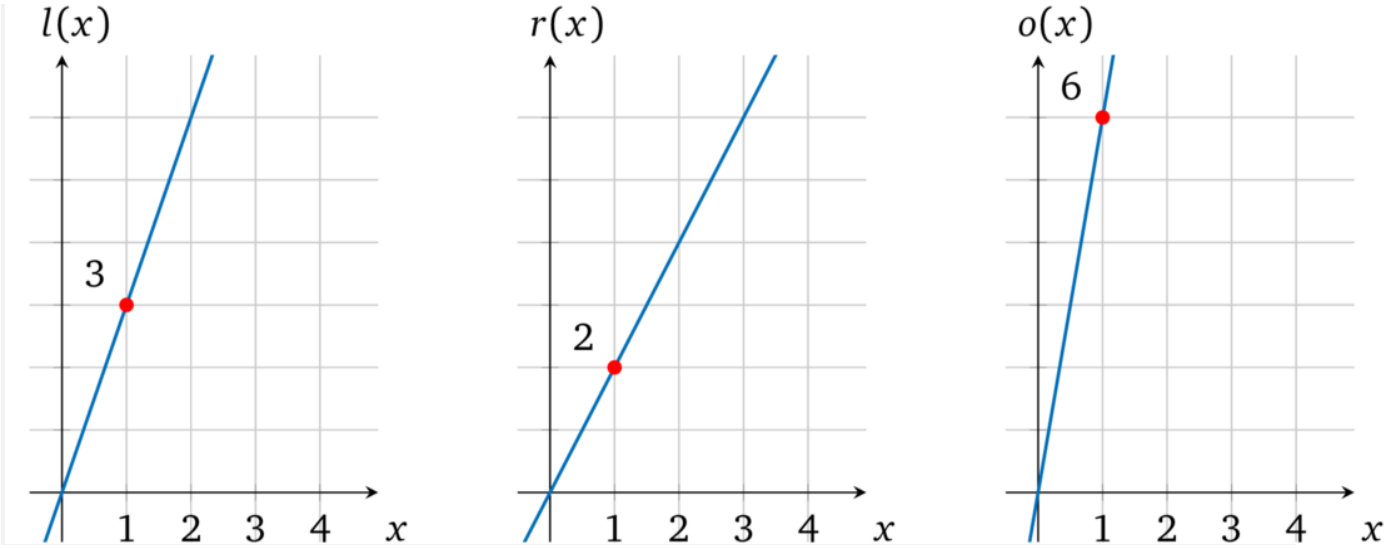
where for some chosen a:

- $l(x)$ — at a represents (evaluates to) the value of the left operand
- $r(x)$ — at a represents the value of the right operand
- $o(x)$ — at a represents the result (output) of the operation

Therefore if the operands and the output are represented correctly for the operation by those polynomials, then the evaluation of $l(a)$ **operator** $r(a) = o(a)$ should hold. And moving *output polynomial* $o(x)$ to the left side of the equation $l(a)$ **operator** $r(a) - o(a) = 0$ is surfacing the fact that the *operation polynomial* $l(x)$ **operator** $r(x) - o(x) = 0$ has to evaluate to 0 at a, if the value represented by the *output polynomial* $o(x)$ is the correct result produced by the **operator** on the values represented by *operand polynomials* $l(x)$ and $r(x)$. Henceforth *operation polynomial* must have the root a if it is valid, and consequently, it must contain cofactor $(x - a)$ as we have established previously (see factorization section), which is the *target polynomial* we prove against, i.e., $t(x) = x - a$.
For example, let us consider operation:

$$3 \times 2 = 6$$

It can be represented by simple polynomials $l(x) = 3x$, $r(x) = 2x$, $o(x) = 6x$, which evaluate to the corresponding values for $a = 1$, i.e., $l(1) = 3$; $r(1) = 2$; $o(1) = 6$.



Note: The value of "a" can be arbitrary.

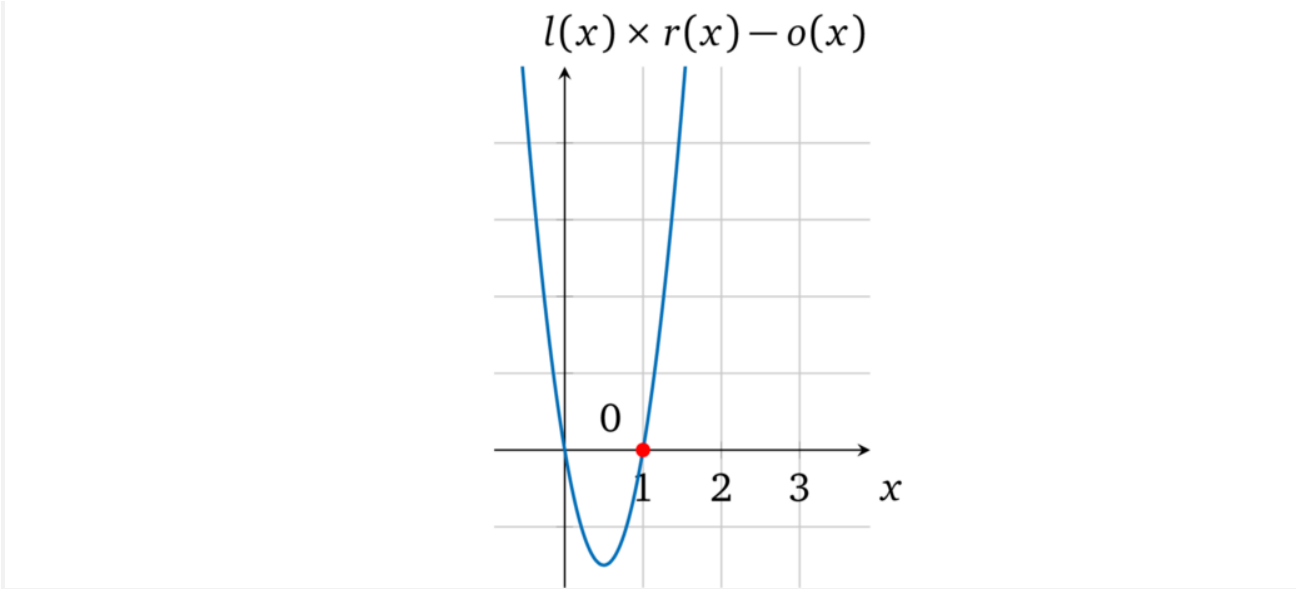
The operation polynomial then will be:

$$l(x) \times r(x) = o(x)$$

$$3x \times 2x = 6x$$

$$6x^2 - 6x = 0$$

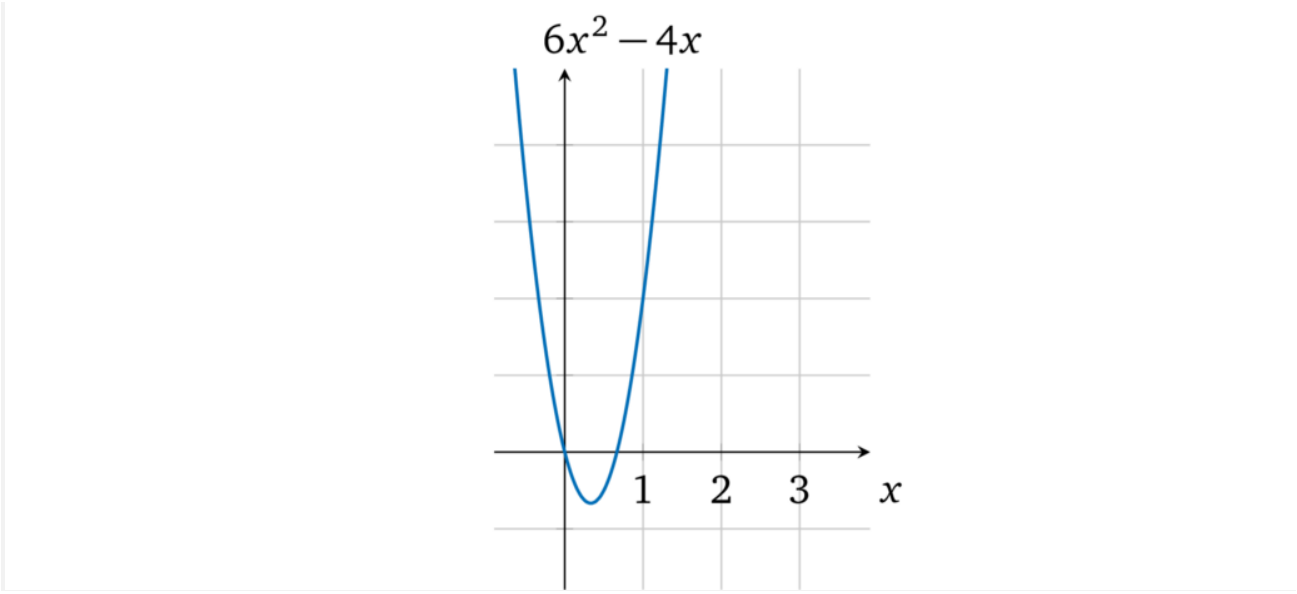
Which is visualised as:



It is noticeable that the operation polynomial has $(x - 1)$ as a co-factor:

$$6x^2 - 6x = 6x(x - 1)$$

Therefore if the prover provides such polynomials $l(x)$, $r(x)$, $o(x)$ instead of former $p(x)$ then the verifier will accept it as valid, since it is divisible by $t(x)$. On the contrary if the prover tries to cheat and substitutes output value with 4, e.g., $o(x) = 4x$, then the *operation polynomial* will be $6x^2 - 4x = 0$:



Which is not have a solution $x = 1$, henceforth $l(x) \times r(x) - o(x)$ is not divisible by $t(x)$ without remainder:

$$h(x) = \begin{array}{r} 6x+2 \\ \hline x-1\,\big)\;\; 6x^2-4x \\ \underline{-6x^2+6x} \\ 2x \\ \underline{-2x+2} \\ 2 \end{array} \quad \Rightarrow \quad h(x) = 6x + 2 + \frac{2}{x-1}$$

Hence such *inconsistent operation* will not be accepted by the verifier (as described in the factorization section).

10.4 Proof of Operation

Let us modify our latest protocol to support a single multiplication operation proof. Recall that previously we had proof of knowledge of polynomial $p(x)$, but now we deal with three $l(x)$, $r(x)$, $o(x)$. While we could define $p(x) = l(x) \times r(x) - o(x)$ there are two counterargument. Firstly, in our protocol, the multiplication of encrypted values (i.e., $l(s) \times r(s)$) is not possible in the proving stage, since pairings can only be used once and it is required for the "polynomial restriction" check. Secondly, this would leave an opportunity for the prover to modify the structure of polynomial at will but still maintain a valid cofactor $t(x)$, for example $p(x) = l(x)$ or $p(x) = l(x) - r(x)$ or even $p(x) = l(x) \times r(x) + o(x)$, as long as $p(x)$ has root a. Such modification effectively means that the proof is about a different statement, which is certainly not desired.

That is why the evaluations of polynomials $l(s)$, $r(s)$, $o(s)$ have to be provided separately by the prover. This means that the *knowledge of polynomial* must be adjusted. In essence what a verifier needs to check in encrypted space is that $l(s) \times r(s) - o(s) = t(s)h(s)$. While a verifier can perform multiplication using cryptographic pairings, the subtraction ($- o(x)$) is an expensive operation (would require to find inverse of $g^{o(s)}$) that is why we move $o(x)$ to the right side of the equation: $l(x)r(x) = t(x)h(x) + o(x)$. In encrypted space verifier's check translates to:

$$e\left(g^{l(s)}, g^{r(s)}\right) = e\left(g^{t(s)}, g^{h(s)}\right) \cdot e\left(g^{o(s)}, g\right)$$
$$e(g,g)^{l(s)r(s)} = e(g,g)^{t(s)h(s)} \cdot e(g,g)^{o(s)}$$
$$e(g,g)^{l(s)r(s)} = e(g,g)^{t(s)h(s)+o(s)}$$

Note: recall that the result of cryptographic pairings supports encrypted addition through multiplication, see section on pairings.

While the *setup* stage stays unchanged, here is the updated protocol:

- Proving
  - assign corresponding coefficients to the $l(x)$, $r(x)$, $o(x)$
  - calculate polynomial $h(x) = \frac{l(x) \times r(x) - o(x)}{t(x)}$
  - evaluate encrypted polynomials $g^{l(s)}$, $g^{r(s)}$, $g^{o(s)}$ and $g^{h(s)}$ using $\left\{g^{s^i}\right\}_{i \in [d]}$
  - evaluate encrypted shifted polynomials $g^{\alpha l(s)}$, $g^{\alpha r(s)}$, $g^{\alpha o(s)}$ using $\left\{g^{\alpha s^i}\right\}_{i \in \{0,\ldots,d\}}$
  - set proof $\pi = \left(g^{l(s)}, g^{r(s)}, g^{o(s)}, g^{h(s)}, g^{\alpha l(s)}, g^{\alpha r(s)}, g^{\alpha o(s)}\right)$
- Verification
  - parse proof π as $\left(g^{l}, g^{r}, g^{o}, g^{h}, g^{l'}, g^{r'}, g^{o'}\right)$
  - polynomial restrictions check:
    $e(g^{l'}, g) = e(g^{l}, g^{\alpha})$
    $e(g^{r'}, g) = e(g^{r}, g^{\alpha})$
    $e(g^{o'}, g) = e(g^{o}, g^{\alpha})$
  - valid operation check: $e\left(g^{l}, g^{r}\right) = e\left(g^{t(s)}, g^{h}\right) \cdot e\left(g^{o}, g\right)$

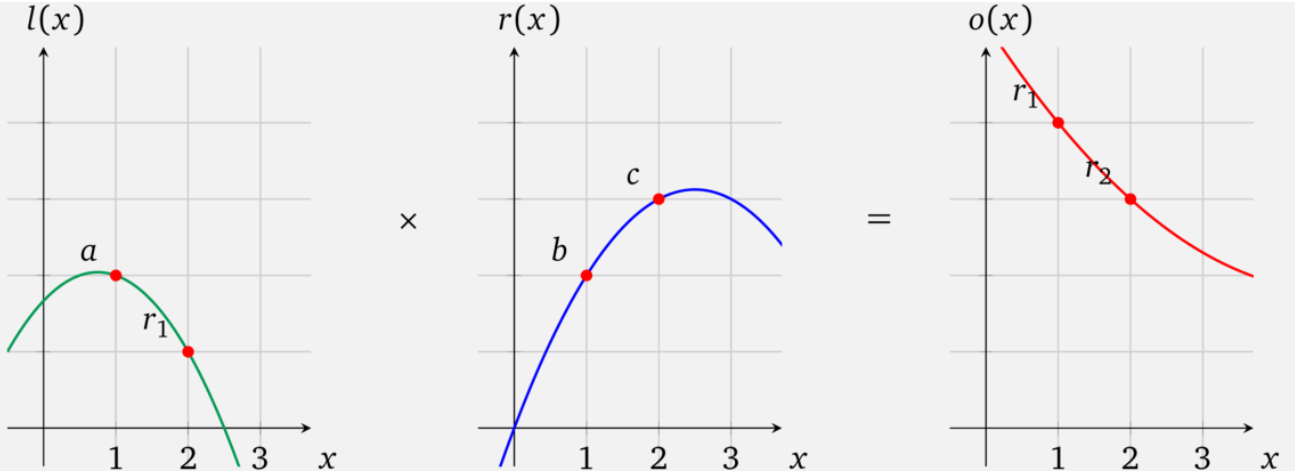
Such protocol allows to prove that the result of multiplication of two values is computed correctly.
One might notice that in the updated protocol we had to let go of the *zero-knowledge* component. The reason for this is to make the transition simpler. We will get back to it in a later section.

10.5 Multiple Operations

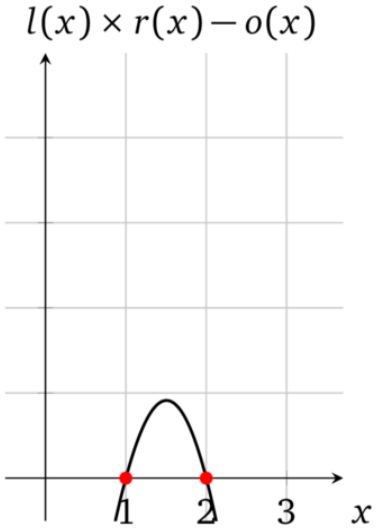
We can prove a single operation, but how do we scale to prove multiple operations (which is our ultimate goal)? Let us try to add just one another operation. Consider the need to compute the product: $a \times b \times c$. In the elemental operation model this would mean two operations:

$$a \times b = r_1$$
$$r_1 \times c = r_2$$

As discussed previously we can represent one such operation by making operand polynomials evaluate to a corresponding value at some arbitrary *x*, for example 1. Having this the properties of polynomials does not restrict us in representing other values at different *x*, for example 2, e.g.:



Such independence allows us to *execute* two operations at once without "mixing" them together, i.e., no interfering. The result of such polynomial arithmetic will be:



Where it is visible that the operation polynomial has roots *x* = 1 and *x* = 2. Therefore both operations are *executed* correctly. Let us have a look at example of 3 multiplications 2 × 1 × 3 × 2, which can be executed as follows:

$$2 \times 1 = 2$$
$$2 \times 3 = 6$$
$$6 \times 2 = 12$$

We need to represent those as operand polynomials, such that for operations represented by $x \in \{1, 2, 3\}$ the $l(x)$ pass correspondingly through 2, 2 and 6, i.e., through points (1, 2), (2, 2), (3, 6), and similarly $r(x) \ni$ (1, 1),(2, 3),(3,2) and $o(x) \ni$ (1, 2), (2, 6), (3, 12).
However, how do we find such polynomials which passes through those points? For any case where we have more than one point, a particular mathematical method has to be used.

10.6 Polynomial Interpolation

In order to construct *operand* and *output polynomials* we need a method which given a set of points produces a *curved* polynomial in such a way that it passes through all those points, it is called *interpolation*. There are different ways available:

- Set of equations with unknowns
- Newton polynomial
- Neville's algorithm
- Lagrange polynomials
- Fast Fourier transform

Let us use the former for example. The idea of such method is that there exists a unique polynomial $p(x)$ of degree at most n with yet *unknown coefficients* which pass through given $n + 1$ points such that for each point $\{(x_i, y_i)\}$, $i \in [n+1]$, the polynomial evaluated at x_i should be equal to y_i, i.e. $p(x_i) = y_i$ for all i. In our case for three points it will be polynomial of degree 2 of the form:

$$ax^2 + bx + c = y$$

Let us *equalize* the evaluated polynomial for each point of the *left operand polynomial* (green) and solve the system of equations by expressing each coefficient in terms of others:

$$\begin{cases} l(1) = 2 \\ l(2) = 2 \\ l(3) = 6 \end{cases} \Rightarrow \begin{cases} a(1)^2 + b \cdot 1 + c = 2 \\ a(2)^2 + b \cdot 2 + c = 2 \\ a(3)^2 + b \cdot 3 + c = 6 \end{cases} \Rightarrow \begin{cases} a + b + c = 2 \\ 4a + 2b + c = 2 \\ 9a + 3b + c = 6 \end{cases} \Rightarrow$$

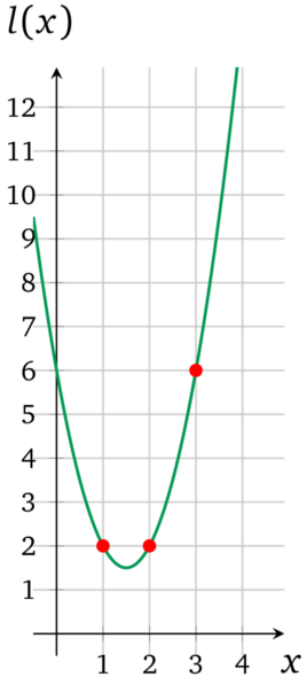
$$\begin{cases} a = 2 - b - c \\ 2b = 2 - 4(2 - b - c) - c \\ c = 6 - 9(2 - b - c) - 3b \end{cases} \Rightarrow \begin{cases} a = 2 - b - c \\ b = \frac{6-3c}{2} \\ c = -12 + 6b + 9c \end{cases} \Rightarrow$$

$$\begin{cases} a = 2 - b - c \\ b = \frac{6-3c}{2} \\ c = -12 + 6(\frac{6-3c}{2}) + 9c \end{cases} \Rightarrow \begin{cases} a = 2 \\ b = -6 \\ c = 6 \end{cases}$$

Therefore the *left operand polynomial* is:

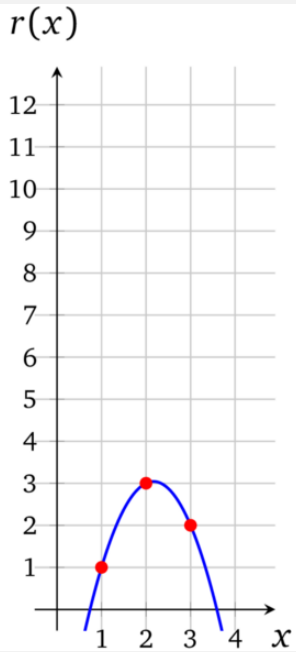
$$l(x) = 2x^2 - 6x + 6$$

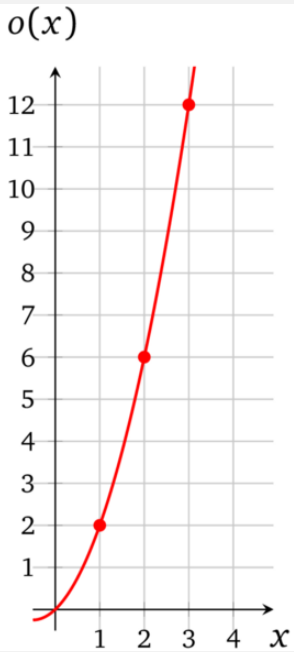
Which corresponds to the following graph:



We can find $r(x)$ and $o(x)$ in the same way:

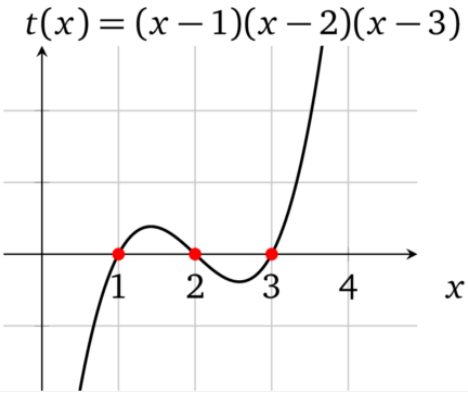
$$r(x) = \frac{-3x^2 + 13x - 8}{2}; \quad o(x) = x^2 + x$$



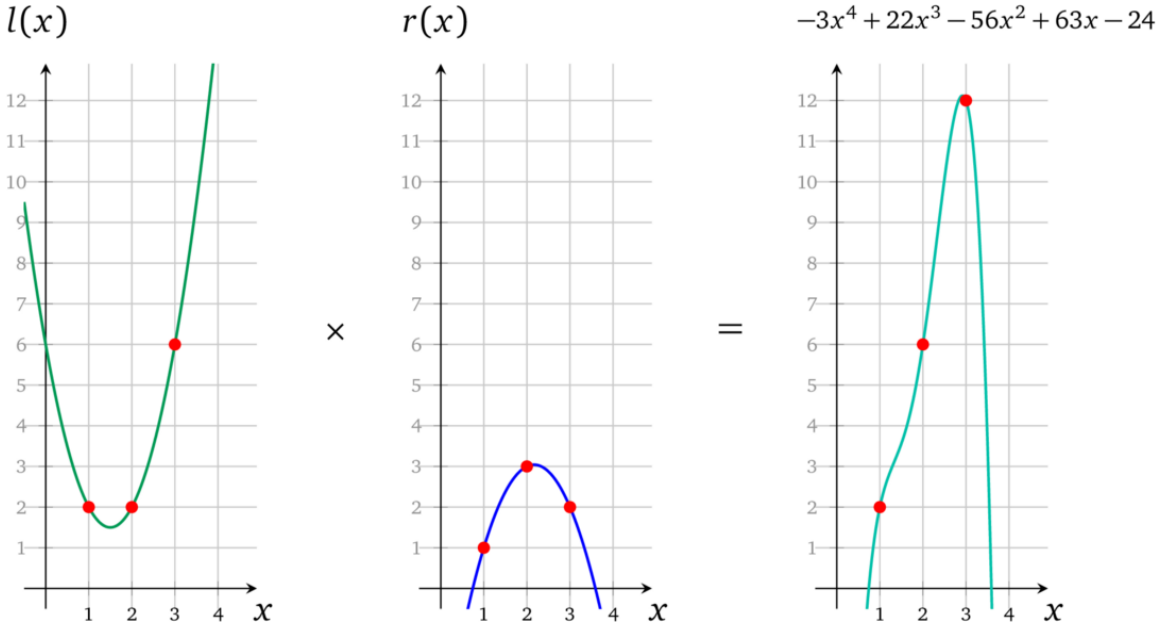


10.7 Multi-Operation Polynomials

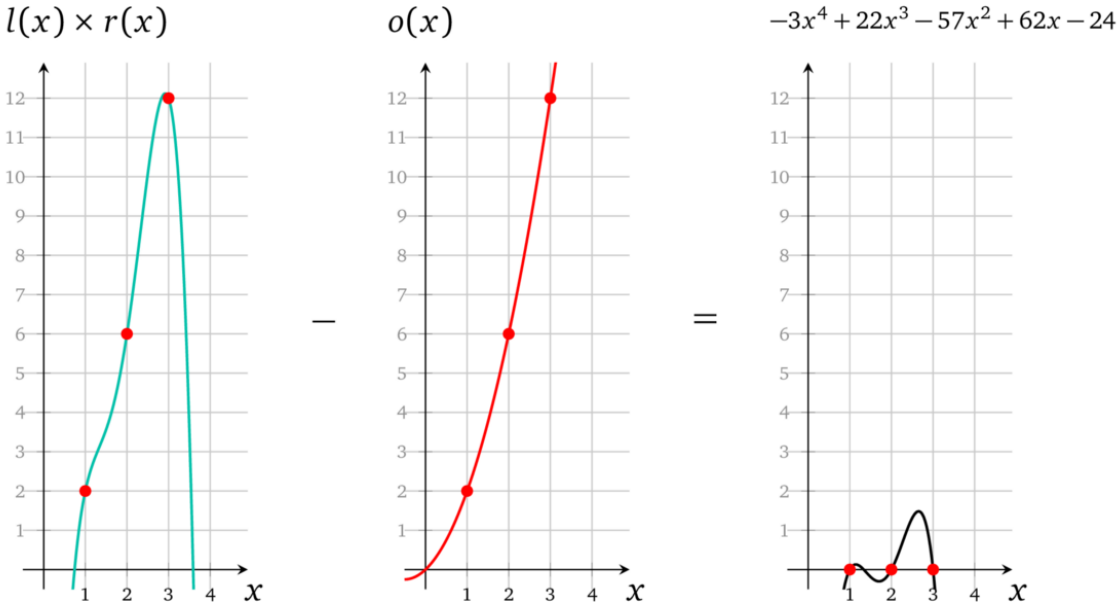
Now we have operand polynomials which represent three operations, let us see step-by-step how the correctness of each operation is verified. Recall that a verifier is looking for equality $l(x) \times r(x) - o(x) = t(x)h(x)$. In this case, because the operations are represented at points $x \in \{1, 2, 3\}$ the target polynomial has to evaluate to 0 at those x-s, in other words, the roots of the $t(x)$ must be 1, 2 and 3, which in elementary form is:



Firstly, *l*(*x*) and *r*(*x*) are multiplied which results in:



Secondly, the *o*(*x*) is subtracted from the result of *l*(*x*) × *r*(*x*):



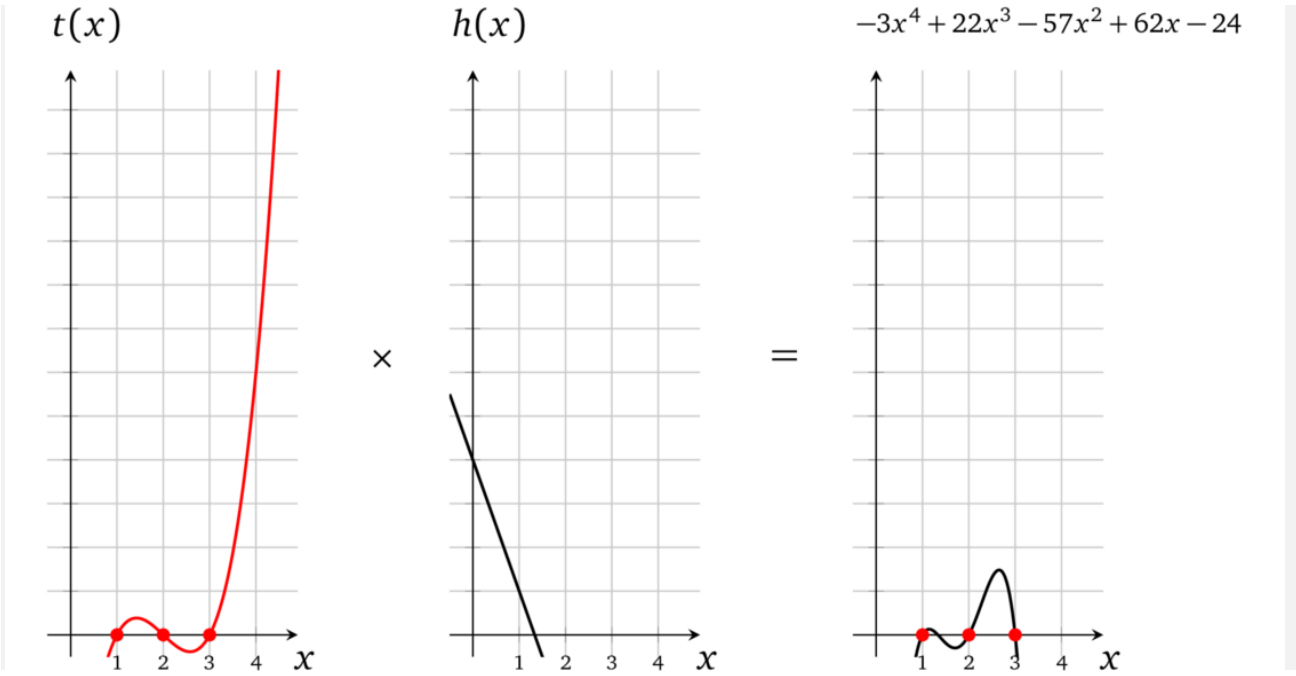
Where it is already visible that every operands multiplication corresponds to a correct result. For the last step a *prover* needs to present a valid cofactor:

$$h(x) = \frac{l(x) \times r(x) - o(x)}{t(x)} = \frac{-3x^4 + 22x^3 - 57x^2 + 62x - 24}{(x-1)(x-2)(x-3)}$$

Using long division we get:

$$\begin{array}{rl} h(x) = & \qquad\qquad -3x \;\; +4 \\ \hline x^3 - 6x^2 + 11x - 6 \big) & -3x^4 + 22x^3 - 57x^2 + 62x - 24 \\ & \;\; 3x^4 - 18x^3 + 33x^2 - 18x \\ \hline & \qquad 4x^3 - 24x^2 + 44x - 24 \\ & \qquad -4x^3 + 24x^2 - 44x + 24 \\ \hline & \qquad\qquad\qquad 0 \end{array}$$

With $h(x) = -3x + 4$ a *verifier* can compute $t(x)h(x)$:

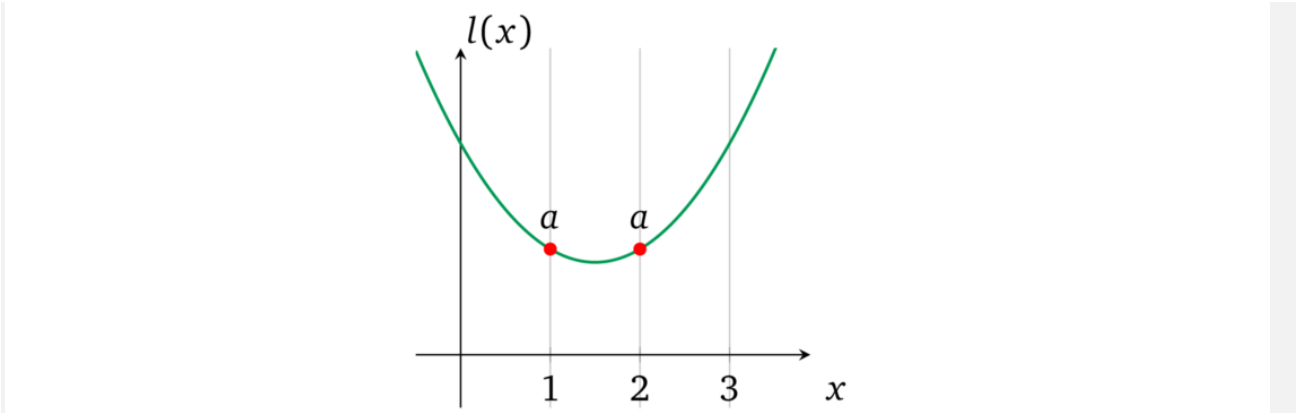


It is now evident that $l(x) \times r(x) - o(x) = t(x)h(x)$ which is what had to be proven.

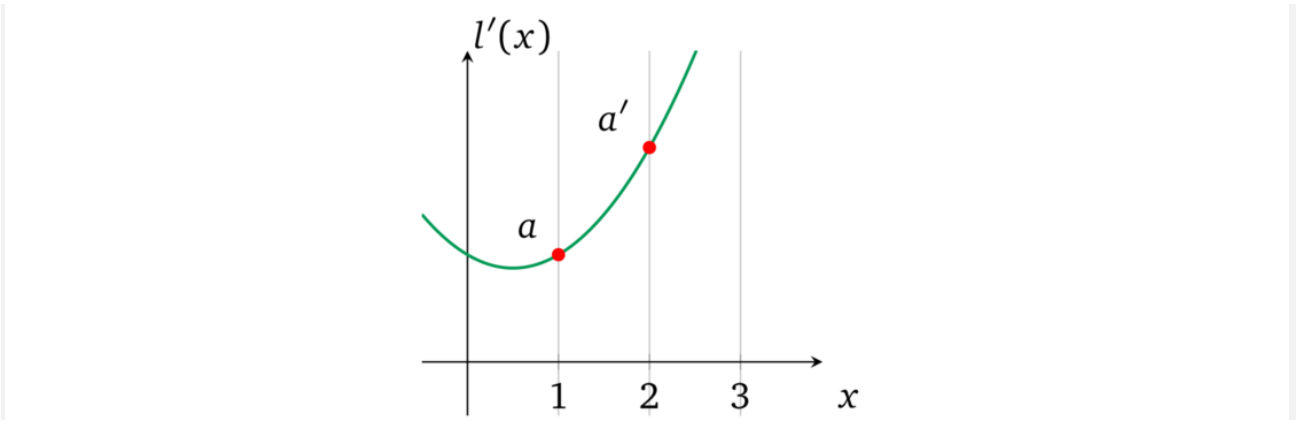
With the multi-operation polynomials approach introduced in part 4, we can prove many operations at once (e.g., millions and more), but there is a critical downside to it.
If the "program," execution for which is being proved, uses the same *variable*, either as an operand or as output, in different operations, for example:

$$\begin{aligned} a \times b &= r_1 \\ a \times c &= r_2 \end{aligned}$$

The *a* will have to be represented in the *left operand polynomial* for both operations as:



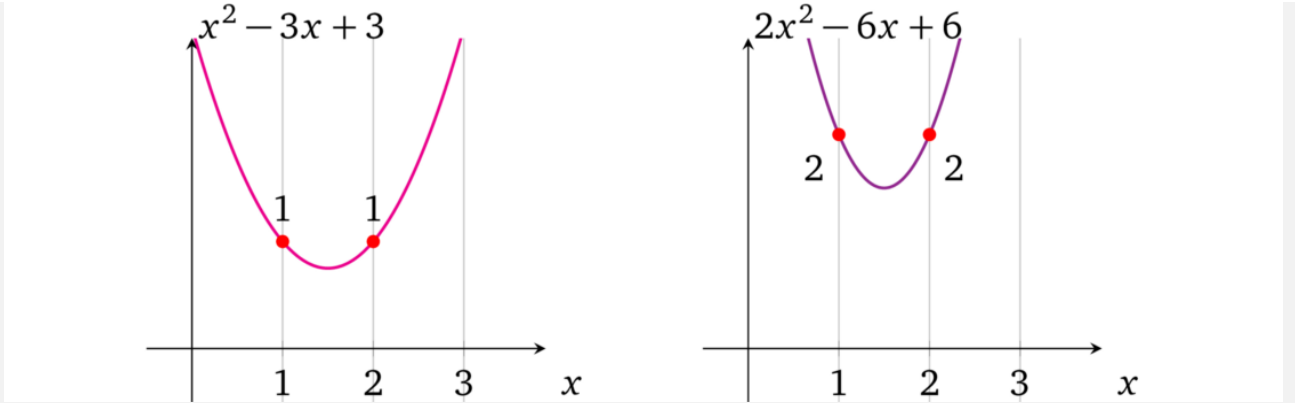
Nevertheless, because our protocol allows prover to set any coefficients to a polynomial, he is not restricted from setting different values of *a* for different operations (i.e., represented by some *x*), e.g.:



This freedom breaks consistency and allows prover to prove the execution of some other program which is not what verifier is interested in. Therefore we must ensure that any variable can only have a single value across every operation it is used in.
Note: variable in this context differs from the regular computer science definition in a sense that it is immutable and is only assigned once per execution.

10.8 Single-Variable Operand Polynomial

Let us consider a simple case (as with the current example) where we have only one variable (e.g., *a*) used in all left operands represented by the *left operand polynomial* *l*(*x*). We have to find out if it is possible to ensure that this polynomial represents the same values of *a* for every operation. The reason why a prover can set different values is that he has control over each coefficient for every exponentiation of *x*. Therefore if those coefficients were constant, that would solve the variability problem. May us have a closer look at polynomials containing equal values. For example examine two polynomials representing equal values for the two operations correspondingly (i.e., at $x = 1$ and $x = 2$), where the first polynomial contains value 1 and the second contains value 2:



Notice that the corresponding coefficients are proportional in each polynomial, such that coefficients in the second are twice as large as in the first, i.e.:

$$2x^2-6x+6=2\times(x^2-3x+3)$$

Therefore when we want to change all the values simultaneously in a polynomial we need to change its proportion, this is due to arithmetic properties of polynomials, if we multiply a polynomial by a number, evaluations at every possible x will be multiplied (i.e., scaled). To verify, try to multiply the first polynomial by 3 or any other number.

Consequently, if a verifier needs to enforce the prover to set the same value in all operations, then it should only be possible to modify the proportion and not the individual coefficients.

So how coefficients proportion can be preserved? We can start by considering what is provided as proof for the *left operand polynomial*. It is an encrypted evaluation of $l(x)$ at some secret s: $g^{l(s)}$, i.e., it is an encrypted number. We already know from the "restricting a polynomial" section how to restrict a verifier to use only the provided exponents of s through an α-shift, such that homomorphic multiplication is the single operation available.

Similarly to restricting a single exponent, the verifier can restrict the whole polynomial at once. Instead of providing separate encryptions and their α-shifts

$$g^{s^1}, g^{s^2}, \ldots, g^{s^d}, \quad g^{\alpha s^1}, g^{\alpha s^2}, \ldots, g^{\alpha s^d}$$

the protocol proceeds:

- Setup
  - construct the respective *operand polynomial* $l(x)$ with corresponding coefficients
  - sample random α and s
  - set proving key with encrypted $l(s)$ and it is "shifted" pair: $\left(g^{l(s)}, g^{\alpha l(s)}\right)$
  - set verification key: (g^{α})
- Proving
  - having operand's value v
    - multiply *operand polynomial*: $\left(g^{l(s)}\right)^{v}$
    - multiply shifted *operand polynomial*: $\left(g^{\alpha l(s)}\right)^{v}$
  - provide *operand polynomial* multiplication proof: $\left(g^{v\,l(s)}, g^{v\,\alpha l(s)}\right)$
- Verification
  - parse the proof as $\left(g^{l}, g^{l'}\right)$
  - verify proportion: $e\left(g^{l'}, g\right) = e\left(g^{l}, g^{\alpha}\right)$

Prover needs to respond with the same α-shift and because he cannot recover α from the proving key the only way to maintain the shift is to multiply both encryptions

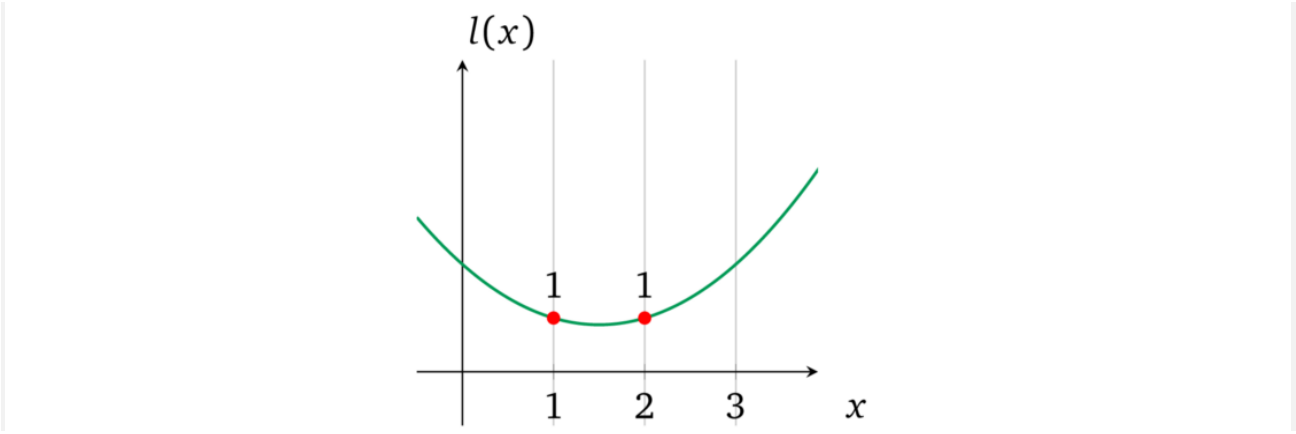
$$g^{l(s)} \quad \text{and} \quad g^{\alpha l(s)}$$

by the same value. Therefore prover cannot modify individual coefficients of $l(x)$, for example if $l(x) = ax^2 + bx + c$ he can only multiply the whole polynomial at once by some value v: $v \cdot (ax^2 + bx + c) = v \cdot ax^2 + v \cdot bx + v \cdot c$. Multiplication by another polynomial is not available since *pairings*, and α-shifts of individual exponents of s are not provided. Prover cannot add or subtract either since:

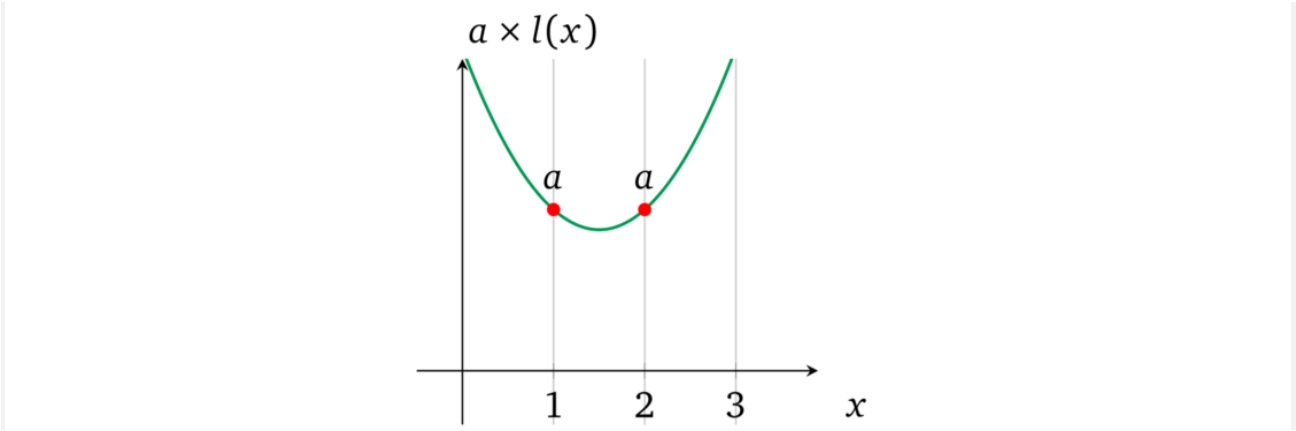
$$g^{\alpha(l(x)+a'x^2+c')} \neq g^{\alpha l(x)} \cdot g^{a'x^2} \cdot g^{c'}$$

This, again, requires the knowledge of unencrypted α

We now have the protocol, but how *operand polynomial* $l(x)$ should be constructed? Since any integer can be derived by multiplying 1, the polynomial should evaluate to 1 for every corresponding operation, e.g.:



This allows a prover to ***assign*** the value of a:



Remark 4.1 *Since verification key contains encyrpted α it is possible to add (or subtract) an arbitrary value v′ to the polynomial, i.e.:*

$$g^{vl(s)} \cdot g^{v'} = g^{vl(s)+v'}$$

$$g^{\alpha vl(s)} \cdot (g^{\alpha})^{v'} = g^{\alpha(vl(s)+v')}$$

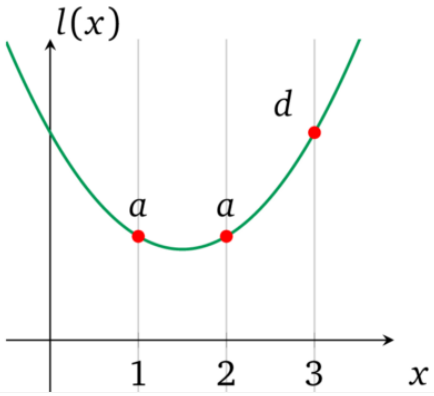
$$e\left(g^{\alpha(vl(s)+v')}, g\right) = e\left(g^{vl(s)+v'}, g^{\alpha}\right)$$

Therefore it is possible to modify the polynomial beyond what is intended by the verifier and prove a different statement. We will address this shortcoming in a further section.

10.9 Multi-Variable Operand Polynomial

We are now able to singularly set value only if all left operands use the same variable. What if we add another one *d*:

$$a \times b = r_1$$
$$a \times c = r_2$$
$$d \times c = r_3$$



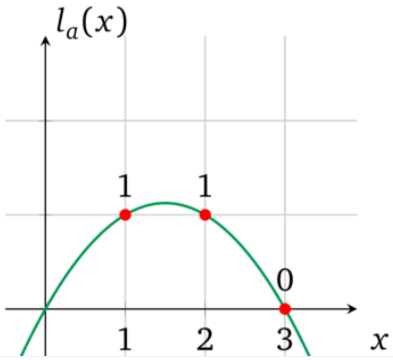
If we have used the same approach we would not be able to set the value separately for each variable, and every distinct variable will be multiplied altogether. Hence such restricted polynomial can support only one *variable*. If we examine properties of polynomials, we will see that adding polynomials together adds distinct evaluations of those polynomials. Therefore we can separate the *operand polynomial* $l(x)$ into *operand variable polynomials*

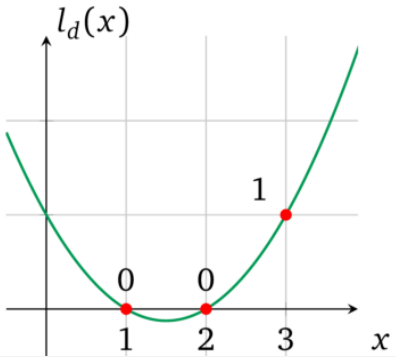
$$l_a(x) \text{ and } l_d(x)$$

(note the subscripts) such that *variables a* and *b* are *assigned* and restricted separately similarly to the previous section and then added together to represent variables of all left operands. Because we add *operand variable polynomials* together, we need to ensure that only one of all the *variables* is represented for each operation by the *operand polynomial*.
Using the arithmetic properties we can construct each *operand variable polynomial* such that if *variable* is used as an operand in the corresponding operation then it evaluates to 1, otherwise to 0. Consecutively 0 multiplied by any value will remain zero and when added together it will be ignored. For our example the variable polynomials must conform to evaluations:

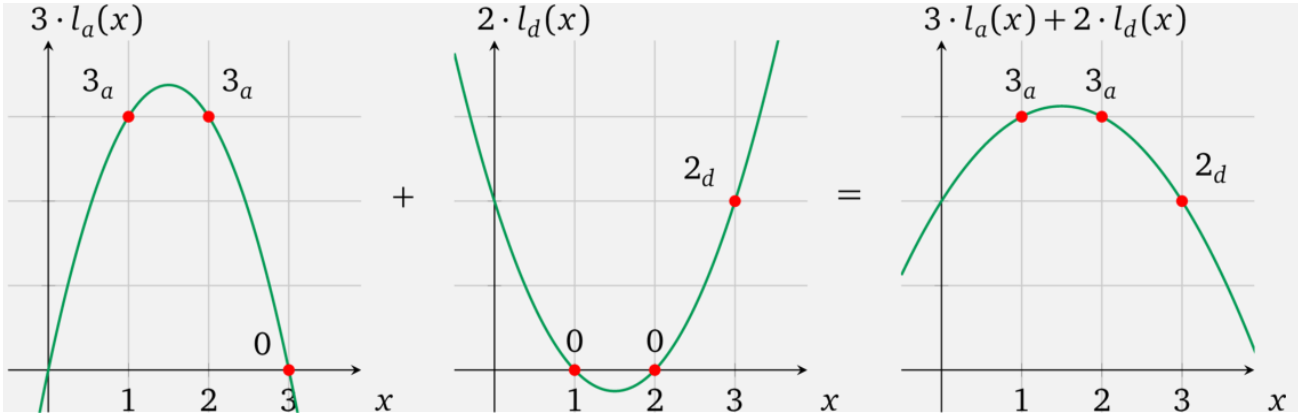
$$l_a(1) = 1,\ l_a(2) = 1 \text{ and } l_a(3) = 0$$
$$l_d(x) = 0,\ l_d(2) = 0 \text{ and } l_d(3) = 1$$

In graph form:





Consequently we can set the value of each variable separately and just add them together to get the operand polynomial, for example if $a = 3$ and $d = 2$:



Note: we are using subscript next to a value to indicate which variable it represents, e.g., 3_a *is a variable a instantiated with value* 3.

Let us denote such composite *operand polynomial* with an upper-case letter from now on, e.g. :

$$L(x) = a\, l_a(x) + d\, l_d(x)$$

and its evaluation value as *L*, i.e., *L* = *L*(*s*). This construction will only be effective if each *operand variable polynomial* is restricted by the verifier, the interaction concerning left operand shall be altered accordingly:

- Setup
  - construct $l_a(x)$, $l_d(x)$ such that it passes through 1 at "operation x" where it is used and through 0 in all other operations
  - sample random s, α
  - evaluate and encrypt *unassigned variable polynomials*:
    $g^{l_a(s)}, g^{l_d(s)}$
  - calculate *shifts* of these polynomials:
    $g^{\alpha l_a(s)}, g^{\alpha l_d(s)}$
  - set proving key:
    $\left(g^{l_a(s)}, g^{l_d(s)}, g^{\alpha l_a(s)}, g^{\alpha l_d(s)}\right)$
  - set verification key:
    (g^{α})
- Proving
  - *assign* values a and d to the variable polynomials:
    $\left(g^{l_a(s)}\right)^a, \left(g^{l_d(s)}\right)^d$
  - *assign* same values to the *shifted* polynomials:
    $\left(g^{\alpha l_a(s)}\right)^a, \left(g^{\alpha l_d(s)}\right)^d$
  - add all *assigned* variable polynomials to form an *operand polynomial*:
    $g^{L(s)} = g^{a l_a(s)} \cdot g^{d l_d(s)} = g^{a l_a(s) + d l_d(s)}$
  - add *shifted assigned variable polynomials* to form a *shifted operand polynomial*:
    $g^{\alpha L(s)} = g^{a\alpha l_a(s)} \cdot g^{d\alpha l_d(s)} = g^{\alpha(a l_a(s) + d l_d(s))}$
  - provide proof of valid assignment of *left operand*:
    $\left(g^{L(s)}, g^{\alpha L(s)}\right)$
- Verification
  - parse proof as $\left(g^{L}, g^{L'}\right)$
  - check that provided polynomials is a sum of multiples of originally provided *unassigned variable polynomials*:

    $e\left(g^{L'}, g\right) = e\left(g^{L}, g^{\alpha}\right)$ which checks that

    $\alpha\, a l_a(s) + \alpha\, d l_d(s) = \alpha \times (a l_a(s) + d l_d(s))$

Note: L(s) and αL(s) represent all variable polynomials at once and since α is used only in evaluation of variable polynomials, the prover has no option but to use provided evaluations and assign same coefficients to original and shifted variable polynomials.

As a consequence the prover:

- is not able to modify provided *variable polynomials* by changing their coefficients, except "assigning" values, because prover is presented only with encrypted evaluations of these polynomials, and because necessary encrypted powers of *s* are unavailable separately with their *α*-shifts
- is not able to add another polynomial to the provided ones because the *α*-ratio will be broken
- is not able to modify operand polynomials through multiplication by some other polynomial *u*(*x*), which could disproportionately modify the values because encrypted multiplication is not possible in pre-pairings space

Note: if we add (or subtract) one polynomial, e.g., $l_a(x)$, to the other, e.g.,

$$l'_d(x) = c_d \cdot l_d(x) + c'_a \cdot l_a(x)$$

that is not really a modification of the polynomial ld(x), but rather changing of the resulting coefficient of the la(x), because they are summed up in the end:

$$L(x) = c_a \cdot l_a(x) + l'_d(x) = \left(c_a + c'_a\right) \cdot l_a(x) + c_d \cdot l_d(x)$$

While the prover restricts the use of polynomials, there is still some freedoms which are not necessary to counteract:

- it is acceptable if the prover decides not to add some of the assigned variable polynomials $l_i(x)$ to form the operand polynomial $L(x)$ because it is the same as to assign the value 0:

$$g^{al_a(x)} = g^{al_a(x)+0l_d(x)}$$

- it is acceptable if the prover adds same variable polynomials multiple times because it is the same as to assign the multiple of that value once, e.g.:

$$g^{al_a(x)} \cdot g^{al_a(x)} \cdot g^{al_a(x)} = g^{3al_a(x)}$$

This approach is applied similarly to the right operand and output polynomials $R(x)$, $O(x)$.

10.10 Constant Coefficients

In the above construction, we have been using evaluations of *unassigned variable polynomials* 1 or 0 as a means to signify if the variable is used in operation or not. Naturally, there is nothing that stops us from using other coefficients as well, including negative ones, because we can *interpolate* polynomials through any necessary points (provided that no two operations occupy same *x*). Examples of such operations are:

$$2a \times 1b = 3r$$

$$-3a \times 1b = -2r$$

Therefore our program can now use constant coefficients, for example:

Algorithm 2: Constant coefficients

```
function calc(w, a, b)
  if w then
    return 3a × b
  else
    return 5a × 2b
```

```
  end if
end function
```

These coefficients will be “hardwired” during the setup stage and similarly to 1 or 0 will be immutable. We can modify the form of operation accordingly:

$$c_a \cdot a \quad \times \quad c_b \cdot b \quad = \quad c_r \cdot r$$

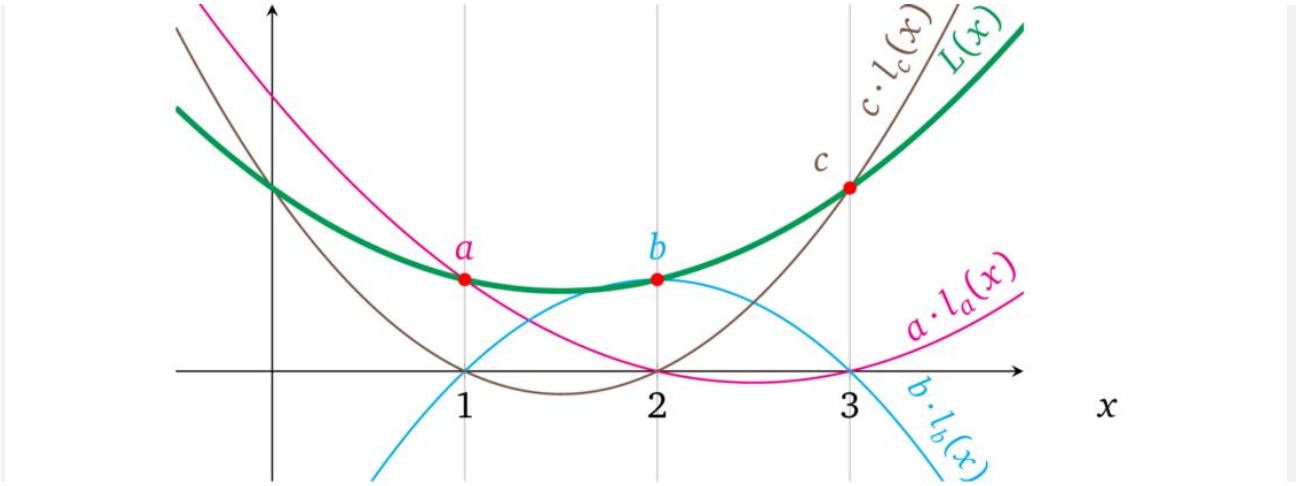
Or more formally, for variables $v_i \in \{v_1, v_2, \ldots, v_n\}$:

$$c_l \cdot v_l \quad \times \quad c_r \cdot v_r \quad = \quad c_o \cdot v_o$$

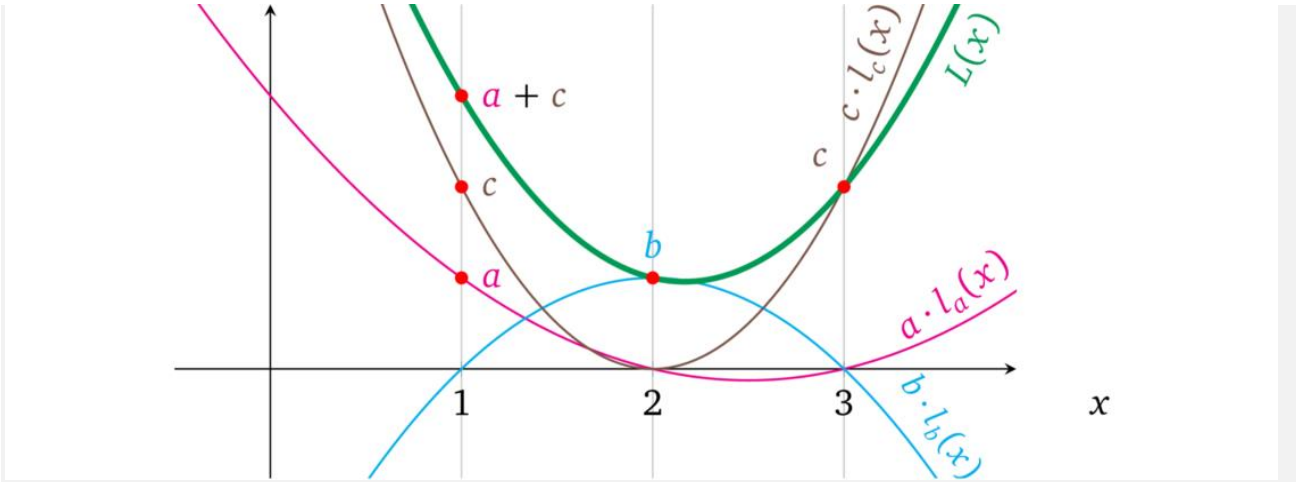
where subscripts *l, r* and *o* are indices of a variable used in operation.
Note: constant coefficient for the same variable can be different in different operations and operands/outputs.

10.11 Addition for Free

Considering the updated construction, it is apparent that in polynomial representation every *operand* expressed by some distinct *x* is a sum of all *operand variable polynomials* such that only single *used* variable can have a non-zero value and all others are zero. The graph demonstrates it best:



We can take advantage of such construction and allow to add any number of necessary *variables* for each operand/output in operation. For example in the first operation, we can add $a + c$ first and only then multiply it by some other operand, e.g., $(a + c) \times b = r$, this can be represented as:



Therefore it is possible to add any number of present variables in a single *operand*, using arbitrary coefficients for each of them, to produce an operand value which will be used in a corresponding operation, as needed in a respective program. Such property effectively allows changing the operation construction to:

$$(c_{l,a} \cdot a + c_{l,b} \cdot b + \ldots) \quad \times \quad (c_{r,a} \cdot a + c_{r,b} \cdot b + \ldots) \quad = \quad (c_{o,a} \cdot a + c_{o,b} \cdot b + \ldots)$$

Or more formally, for variables $v_i \in \{v_1, v_2, \ldots, v_n\}$ and operand variable coefficients:

$$c_{l,i} \in \{c_{l,1}, c_{l,2}, \ldots, c_{r,n}\},\ c_{r,i} \in \{c_{r,1}, c_{r,2}, \ldots, c_{r,n}\},\ c_{o,i} \in \{c_{o,1}, c_{o,2}, \ldots, c_{o,n}\}$$

the construction is:

$$\sum_{i=1}^{n} c_{l,i} \cdot v_i \quad \times \quad \sum_{i=1}^{n} c_{r,i} \cdot v_i \quad = \quad \sum_{i=1}^{n} c_{o,i} \cdot v_i$$

***Note**: each operation's operand has its own set of coefficients c.*

10.12 Addition, Subtraction and Division

We have been focusing on multiplication operation primarily until now. However, in order to be able to execute general computations, a real-life program will also require addition, division, and subtraction.

Addition In previous section we have established that we can add variables in context of a single operand, which is then multiplied by another operand, e.g., $(3a + b) \times d = r$, but what if we need just addition without multiplication, for example, if a program needs to compute a + b, we can express this as:

$$(a + b) \quad \times \quad 1 \quad = \quad r$$

***Note**: because our construction requires both a constant coefficient and a variable ($c \cdot v$) for every operand, the value of 1 is expressed as $c_1 \cdot v_1$, and while $c_1 = 1$ can be "hardwired" into a corresponding polynomial, the v_1 is a variable and can be assigned any value, therefore we must enforce the value of v_1 through constraints as explained in a further section.*

Subtraction Subtraction is almost identical to addition, the only difference is a negative coefficient, e.g., for a – b:

$$(a + -1 \cdot b) \quad \times \quad 1 \quad = \quad r$$

Division If we examine the division operation

$$\frac{\text{factor}}{\text{divisor}} = \text{result}$$

we would see that the result of the division is the number we need to multiply divisor by to produce the factor. Therefore we can express the same meaning through multiplication: divisor x result = factor. Consequently, if we want to prove the division operation a / b= r , it can be expressed as:

$$b \quad \times \quad r \quad = \quad a$$

***Note**: the operation's construction is also called "constraint" because the operation represented by polynomial construction does not compute results per se, but rather checks that the prover already knows variables (including result), and they are valid for the operation, i.e., the prover is constrained to provide consistent values no matter what they are.*

***Note**: all those arithmetic operations were already present; therefore modification of the operation's construction is not needed.*

10.13 Example Computation

Having the general operation's construction, we can convert our original algorithm 1 into a set of operations and further into polynomial form. Let us consider the mathematical form of the algorithm (we will use variable *v* to capture the result of evaluation):

$$w \times (a \times b) + (1 - w) \times (a + b) = v$$

It has three multiplications, and because the operation construction supports only one, there will be at least 3 operations. However, we can simplify the equation:

$$w \times (a \times b) + a + b - w \times (a + b) = v$$

$$w \times (a \times b - a - b) = v - a - b$$

Now it requires two multiplications while maintaining same relationships. In complete form the operations are:

$$1: \quad 1 \cdot a \times 1 \cdot b = 1 \cdot m$$

$$2: \quad 1 \cdot w \times 1 \cdot m + -1 \cdot a + -1 \cdot b = 1 \cdot v + -1 \cdot a + -1 \cdot b$$

We can also add a constraint that requires *w* to be binary, otherwise a prover can use any value for *w* rendering computation incorrect:

$$3: \quad 1 \cdot w \times 1 \cdot w = 1 \cdot w$$

To see why *w* can only be 0 or 1, we can represent the equation as $w^2 - w = 0$ and further as $(w - 0)(w - 1) = 0$ where 0 and 1 are the only solutions.
These totals to 5 variables, with 2 in the left operand, 4 in the right operand and 5 in the output. The operand polynomials are:

$$L(x) = a \cdot l_a(x) + w \cdot l_w(x)$$

$$R(x) = m \cdot r_m(x) + a \cdot r_a(x) + b \cdot r_b(x)$$

$$O(x) = m \cdot o_m(x) + v \cdot o_v(x) + a \cdot o_a(x) + b \cdot o_b(x)$$

where each *variable polynomial* must evaluate to a corresponding coefficient for each of 3 operations or to 0 if the variable isn't present in the operation's operand or output:

$$l_a(1) = 1;\ l_a(2) = 0;\ l_a(3) = 0;\ r_m(1) = 0;\ r_m(2) = 1;\ r_m(3) = 0;\ o_m(1) = 1;\ o_m(2) = 0;\ o_m(3) = 0;$$

$$l_w(1) = 0;\ l_w(2) = 1;\ l_w(3) = 1;\ r_a(1) = 0;\ r_a(2) = -1;\ r_a(3) = 0;\ o_v(1) = 0;\ o_v(2) = 1;\ o_v(3) = 0;$$

$$r_b(1) = 1;\ r_b(2) = -1;\ r_b(3) = 0;\ o_a(1) = 0;\ o_a(2) = -1;\ o_a(3) = 0;$$

$$r_w(1) = 0;\ r_w(2) = 0;\ r_w(3) = 1;\ o_b(1) = 0;\ o_b(2) = -1;\ o_b(3) = 0;$$

$$o_w(1) = 0;\ o_w(2) = 0;\ o_w(3) = 1;$$

Consequently the cofactor polynomial is $t(x) = (x - 1)(x - 2)(x - 3)$, which will ensure that all three operations are computed correctly.

Next we leverage polynomial interpolation to find each *variable polynomial*:

$$l_a(x) = \frac{1}{2}x^2 - \frac{5}{2}x + 3; \qquad r_m(x) = -x^2 + 4x - 3; \qquad o_m(x) = \frac{1}{2}x^2 - \frac{5}{2}x + 3;$$

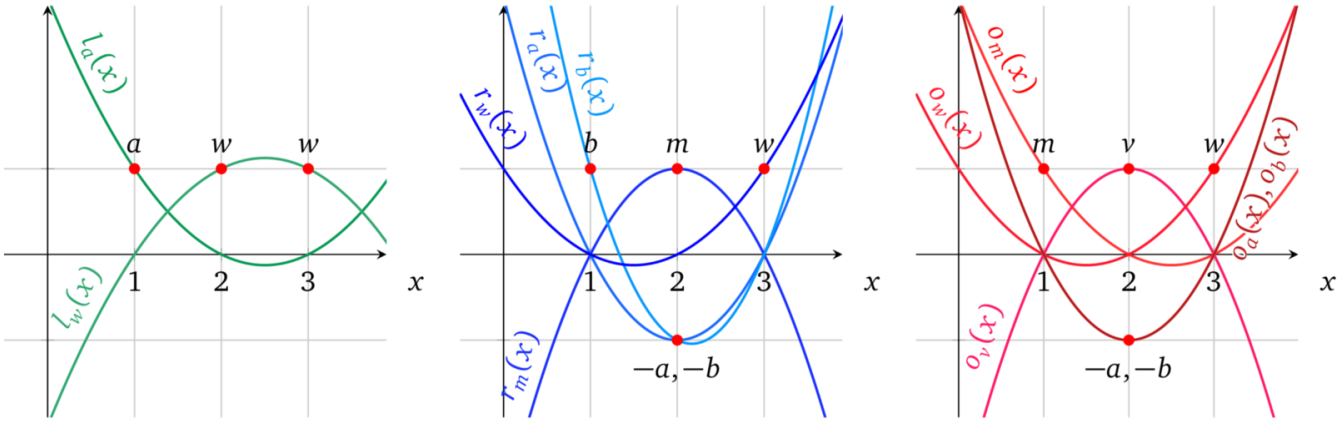
$$l_w(x) = -\frac{1}{2}x^2 + \frac{5}{2}x - 2; \qquad r_a(x) = x^2 - 4x + 3; \qquad o_v(x) = -x^2 + 4x - 3;$$

$$r_b(x) = \frac{3}{2}x^2 - \frac{13}{2}x + 6; \qquad o_a(x) = x^2 - 4x + 3;$$

$$r_w(x) = \frac{1}{2}x^2 - \frac{3}{2}x + 1; \qquad o_b(x) = x^2 - 4x + 3;$$

$$o_w(x) = \frac{1}{2}x^2 - \frac{3}{2}x + 1;$$

Which are plotted as:



We are ready to prove computation through polynomials. Firstly, let us choose input values for the function, for example $w = 1$, $a = 3$, $b = 2$. Secondly, calculate values of intermediary variables from operations:

$$m = a \times b = 6$$

$$v = w(m - a - b) + a + b = 6$$

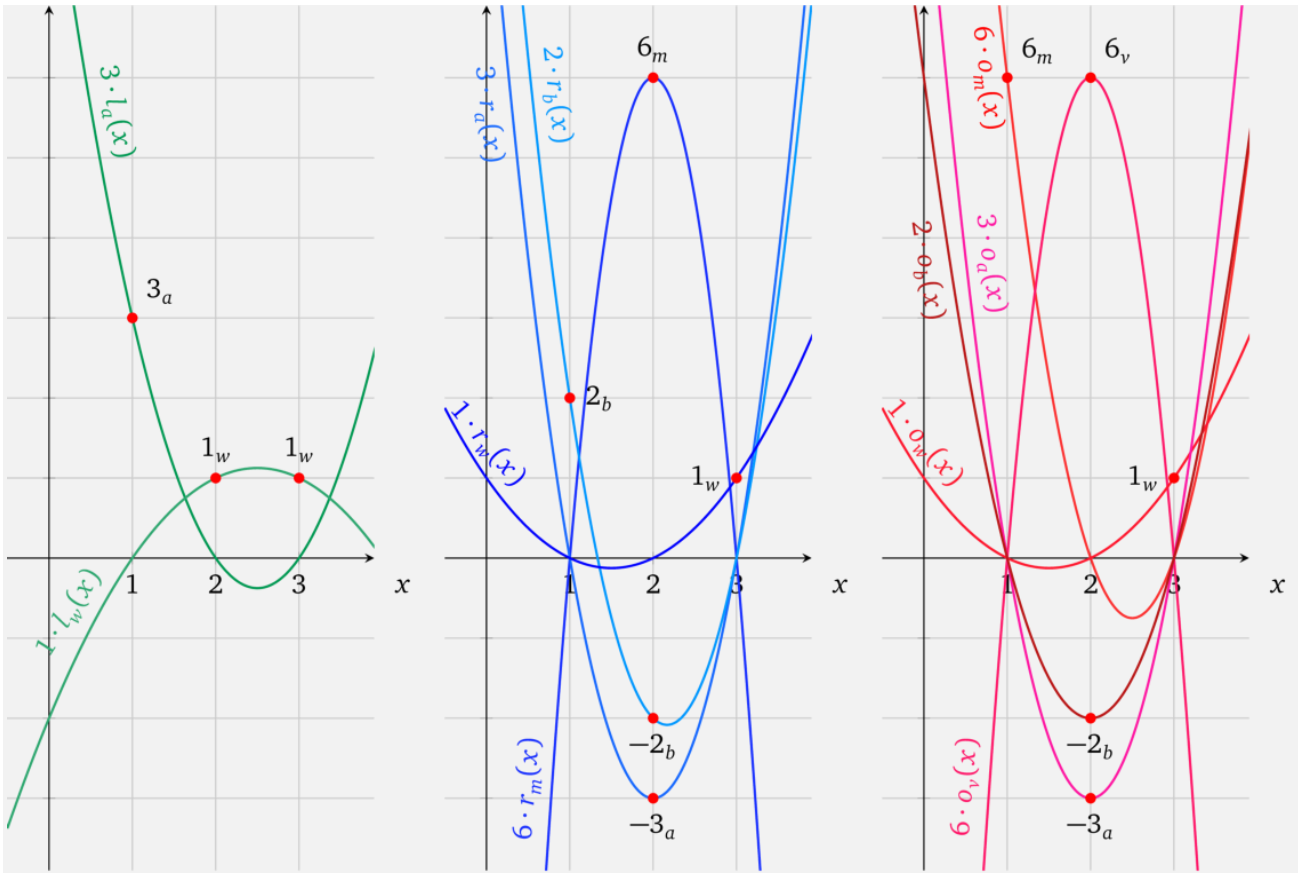
After, we assign all values involved in the computation of the result to the corresponding *variable polynomials* and sum them up to form operand and output polynomials:

$$L(x) = 3 \cdot l_a(x) + 1 \cdot l_w(x) = x^2 - 5x + 7$$

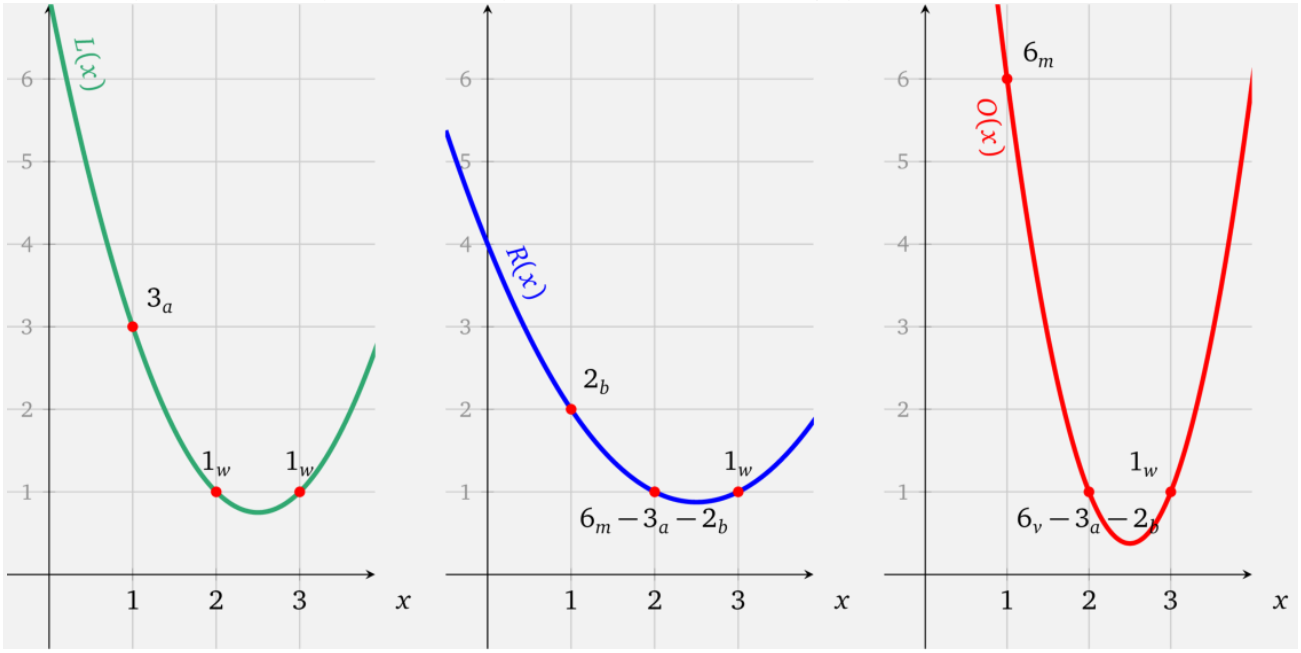
$$R(x) = 6 \cdot r_m(x) + 3 \cdot r_a(x) + 2 \cdot r_b(x) + 1 \cdot r_w(x) = \frac{1}{2}x^2 - 2\frac{1}{2}x + 4$$

$$O(x) = 6 \cdot o_m(x) + 6 \cdot o_v(x) + 3 \cdot o_a(x) + 2 \cdot o_b(x) + 1 \cdot o_w(x) = 2\frac{1}{2}x^2 - 12\frac{1}{2}x + 16$$

and in the graph form these are:



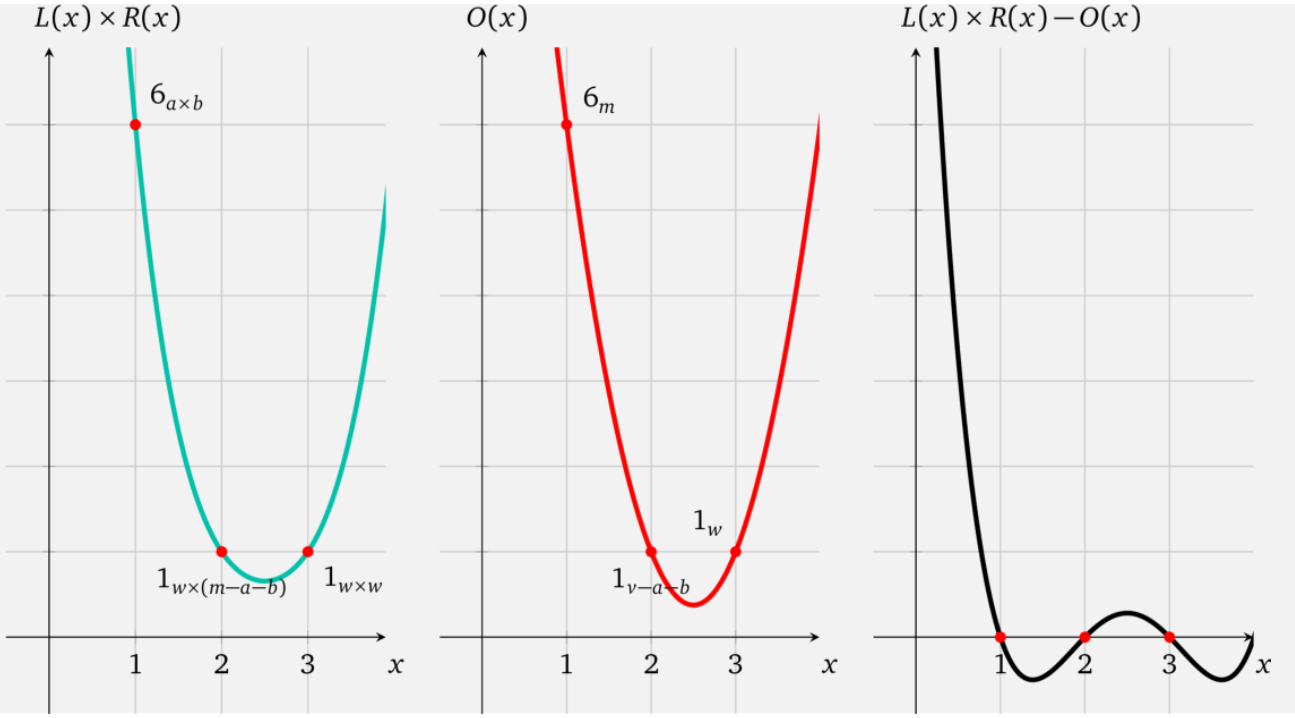
Summed up to represent operand and output values in corresponding operations:



We need to prove that $L(x) \times R(x) - O(x) = t(x)h(x)$, therefore we find $h(x)$:

$$h(x) = \frac{L(x) \times R(x) - O(x)}{t(x)} = \frac{\frac{1}{2}x^4 - 5x^3 + \frac{35}{2}x^2 - 25x + 12}{(x-1)(x-2)(x-3)} = \frac{1}{2}x - 2$$

In a graph form it is represented as:



Where it's visible that polynomial $L(x) \times R(x) - O(x)$ has solutions $x = 1$, $x = 2$ and $x = 3$, and therefore $t(x)$ is its cofactor, which would not be the case if we used inconsistent values of variables.
That is how the knowledge of variable values for a correct computation execution is proven on the level of polynomials. A prover is then proceeding with a cryptographic portion of the protocol.

We went through many important modifications of the knowledge of polynomial protocol to make it general-purpose, so let us see how it is defined now. Assuming agreed upon function $f(*)$ the result of computation of which is the subject of the proof, with the number of operations d, the number of variables n and corresponding to them coefficients

$$\left\{c_{\mathrm{L},i,j}, c_{\mathrm{R},i,j}, c_{\mathrm{O},i,j}\right\}_{i \in \{1,\ldots,n\}, j \in \{1,\ldots,d\}}:$$

- Setup
  - construct *variable polynomials* for left operand $\{l_i(x)\}_{i\in\{1,\dots,n\}}$ such that for all operations $j \in \{1,\dots,d\}$ they evaluate to corresponding coefficients, i.e., $l_i(j) = c_{\mathrm{L},i,j}$, and similarly for right operand and output
  - sample random s, α
  - calculate $t(x) = (x-1)(x-2)\dots(x-d)$ and its evaluation $g^{t(s)}$
  - compute proving key:
    $\left(\left\{g^{s^k}\right\}_{k\in[d]}, \left\{g^{l_i(s)}, g^{r_i(s)}, g^{o_i(s)}, g^{\alpha l_i(s)}, g^{\alpha r_i(s)}, g^{\alpha o_i(s)}\right\}_{i\in\{1,\dots,n\}} \right)$
  - compute verification key:
    $\left(g^{t(s)}, g^{\alpha}\right)$
- Proving
  - compute function $f(*)$ and therefore corresponding variables values $\{v_i\}_{i\in\{1,\dots,n\}}$
  - calculate $h(x) = \frac{L(x)\times R(x) - O(x)}{t(x)}$, where $L(x) = \sum_{i=1}^{n} v_i \cdot l_i(x)$, and similarly $R(x), O(x)$
  - assign variable values and *sum up* to get operand polynomials:
    $g^{L(s)} = \left(g^{l_1(s)}\right)^{v_1} \cdots \left(g^{l_n(s)}\right)^{v_n}, \; g^{R(s)} = \prod_{i=1}^{n} \left(g^{r_i(s)}\right)^{v_i}, \; g^{O(s)} = \prod_{i=1}^{n} \left(g^{o_i(s)}\right)^{v_i}$
  - assign variable values to the shifted polynomials:
    $g^{\alpha L(s)} = \prod_{i=1}^{n} \left(g^{\alpha l_i(s)}\right)^{v_i}, \; g^{\alpha R(s)} = \prod_{i=1}^{n} \left(g^{\alpha r_i(s)}\right)^{v_i}, \; g^{\alpha O(s)} = \prod_{i=1}^{n} \left(g^{\alpha o_i(s)}\right)^{v_i}$
  - calculate encrypted evaluation $g^{h(s)}$ using provided powers of s: $\left\{g^{s^k}\right\}_{k\in[d]}$
  - set proof: $\left(g^{L(s)}, g^{R(s)}, g^{O(s)}, g^{\alpha L(s)}, g^{\alpha R(s)}, g^{\alpha O(s)}, g^{h(s)}\right)$
- Verification
  - parse proof as $\left(g^{L}, g^{R}, g^{O}, g^{L'}, g^{R'}, g^{O'}, g^{h}\right)$
  - variable polynomials restriction check:
    $e(g^{L}, g^{\alpha}) = e(g^{L'}, g), \quad e(g^{R}, g^{\alpha}) = e(g^{R'}, g), \quad e(g^{O}, g^{\alpha}) = e(g^{O'}, g)$
  - valid operations check:
    $e(g^{L}, g^{R}) = e(g^{t}, g^{h}) \cdot e(g^{O}, g)$

Note: using symbol ∏ allows for a concise way to express product of multiple elements, e.g.:

$$\prod_{i=1}^{n} v_i = v_1 \cdot v_2 \cdot \ldots \cdot v_n$$

The set of all the variable polynomials $\{l_i(x), r_i(x), o_i(x)\}$ for $i \in \{1, \ldots, n\}$ and the target polynomial $t(x)$ is called a *quadratic arithmetic program* (QAP, introduced in [Gen+12]).
While the protocol is sufficiently robust to allow a general computation verification, there are two security considerations that must be addressed.

10.14 Non-Interchangeability of Operands and Output

Because we use the same α for all operands of *variable polynomials restriction check* there is nothing that prevents prover from:

- using variable polynomials from other operands, e.g., $L'(s) = o_1(s) + r_1(s) + r_1(s) + \ldots$
- swapping *operand polynomials* completely, e.g., $O(s)$ with $L(s)$ will result in operation $O(s) \times R(s) = L(s)$
- re-using same operand polynomials e.g., $L(s) \times L(s) = O(s)$

This interchangeability means that the prover can alter the execution and effectively prove some other computation. The obvious way to prevent such behavior is to use different α-s for the different operands, concretely we modify:

- Setup
  - ...
  - sample random $\alpha_l, \alpha_r, \alpha_o$ instead of α
  - calculate corresponding "shifts" $\left\{g^{\alpha_l l_i(s)}, g^{\alpha_r r_i(s)}, g^{\alpha_o o_i(s)}\right\}_{i \in \{1 \ldots n\}}$
  - proving key: $\left(\left\{g^{s^k}\right\}_{k \in [d]}, \left\{g^{l_i(s)}, g^{r_i(s)}, g^{o_i(s)}, g^{\alpha_l l_i(s)}, g^{\alpha_r r_i(s)}, g^{\alpha_o o_i(s)}\right\}_{i \in \{1 \ldots n\}}\right)$
  - verification key: $\left(g^{t(s)}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}\right)$
- Proving
  - ...
  - assign variables to the "shifted" polynomials

    $g^{\alpha_l L(s)} = \prod_{i=1}^{n} \left(g^{\alpha_l l_i(s)}\right)^{v_i}$, $g^{\alpha_r R(s)} = \prod_{i=1}^{n} \left(g^{\alpha_r r_i(s)}\right)^{v_i}$, $g^{\alpha_o O(s)} = \prod_{i=1}^{n} \left(g^{\alpha_o o_i(s)}\right)^{v_i}$
  - set proof: $\left(g^{L(s)}, g^{R(s)}, g^{O(s)}, g^{\alpha_l L(s)}, g^{\alpha_r R(s)}, g^{\alpha_o O(s)}, g^{h(s)}\right)$
- Verification
  - ...
  - variable polynomials restriction check:

    $e\left(g^{L}, g^{\alpha_l}\right) = e\left(g^{L'}, g\right), \quad e\left(g^{R}, g^{\alpha_r}\right) = e\left(g^{R'}, g\right), \quad e\left(g^{O}, g^{\alpha_o}\right) = e\left(g^{O'}, g\right)$

It is now not possible to use variable polynomials from other operands since following α-s are not known to the prover:

$$\alpha_l, \alpha_r, \alpha_o$$

10.15 Variable Consistency Across Operands

For any variable v_i we have to *assign* its value to a *variable polynomial* for each corresponding operand, i.e.:

$$\left(g^{l_i(s)}\right)^{v_i}, \left(g^{r_i(s)}\right)^{v_i}, \left(g^{o_i(s)}\right)^{v_i}$$

Because the validity of each of the *operand polynomials* is checked separately, no enforcement requires to use same variable values in the corresponding *variable polynomials*. This means that the value of variable v_1 in left operand can differ from variable v_1 in the right operand or the output.
We can enforce equality of a variable value across operands through already familiar approach of restricting a polynomial (as we did with variable polynomials). If we can create a "shifted checksum" variable polynomial across all operands, that would restrain prover such that he can assign only same value. A verifier can combine polynomials for each variable into one, e.g.,

$$g^{l_i(s)+r_i(s)+o_i(s)}$$

and shift it by some other random value β, i.e.,

$$g^{\beta(l_i(s)+r_i(s)+o_i(s))}$$

This shifted polynomials are provided to the prover to assign values of the variables alongside with variable polynomials:

$$\left(g^{l_i(s)}\right)^{v_{\mathrm{L},i}}, \left(g^{r_i(s)}\right)^{v_{\mathrm{R},i}}, \left(g^{o_i(s)}\right)^{v_{\mathrm{O},i}}, \left(g^{\beta(l_i(s)+r_i(s)+o_i(s))}\right)^{v_{\beta,i}}$$

And the β is encrypted and added to the verification key g^{β}. Now, if the values of all v_i were the same, i.e.,

$$v_{\mathrm{L},i} = v_{\mathrm{R},i} = v_{\mathrm{O},i} = v_{\beta,i} \quad \text{for } i \in \{1, \ldots, n\}$$

the equation shall hold:

$$e\left(g^{v_{\mathrm{L},i} \cdot l_i(s)} \cdot g^{v_{\mathrm{R},i} \cdot r_i(s)} \cdot g^{v_{\mathrm{O},i} \cdot o_i(s)}, g^{\beta}\right) = e\left(g^{v_{\beta,i} \cdot \beta(l_i(s)+r_i(s)+o_i(s))}, g\right)$$

While this is a useful consistency check, due to the non-negligible probability that at least two of $l(s)$, $r(s)$, $o(s)$ could either have same evaluation value or one polynomial is divisible by another etc., this would allow the prover to factor values such that at least two of them are non-equal but the equation holds, rendering the check ineffective:

$$\left(v_{\mathrm{L},i} \cdot l_i(s) + v_{\mathrm{R},i} \cdot r_i(s) + v_{\mathrm{O},i} \cdot o_i(s)\right) \cdot \beta = v_{\beta,i} \cdot \beta \cdot (l_i(s) + r_i(s) + o_i(s))$$

For example, let us consider a single operation, where it is the case that $l(x) = r(x)$. We will denote evaluation of those two as $w = l(s) = r(s)$ and the $y = o(s)$. The equation then will look as:

$$\beta(v_{\mathrm{L}}\, w + v_{\mathrm{R}}\, w + v_{\mathrm{O}}\, y) = v_{\beta} \cdot \beta(w + w + y)$$

Such form allows, for some arbitrary v_R and v_O, to set $v_\beta = v_O$, $v_L = 2v_O - v_R$, which will translate into:

$$\beta(2v_O\, w - v_R\, w + v_R\, w + v_O\, y) = v_O \cdot \beta(2w + y)$$

Hence such consistency strategy is not effective. A way to mitigate this is to use different β for each operand, ensuring that operand's *variable polynomials* will have unpredictable values. Following are the protocol modifications:

- Setup
  - … sample random $\beta_l, \beta_r, \beta_o$
  - calculate, encrypt and add to the proving key the *variable consistency polynomials*:
    $\left\{g^{\beta_l l_i(s)+\beta_r r_i(s)+\beta_o o_i(s)}\right\}_{i\in\{1,\ldots,n\}}$
  - encrypt β-s and add to the verification key: $\left(g^{\beta_l}, g^{\beta_r}, g^{\beta_o}\right)$
- Proving
  - … assign variable values to the *variable consistency polynomials*:
    $g^{z_i(s)} = \left(g^{\beta_l l_i(s)+\beta_r r_i(s)+\beta_o o_i(s)}\right)^{v_i}$ for $i \in \{1, \ldots, n\}$
  - add assigned polynomials in encrypted space:
    $$g^{Z(s)} = \prod_{i=1}^{n} g^{z_i(s)} = g^{\beta_l L(s)+\beta_r R(s)+\beta_o O(s)}$$
  - add to the proof: $g^{Z(s)}$
- Verification
  - … check the consistency between provided *operand polynomials* and the "checksum" polynomial:
    $e\left(g^L, g^{\beta_l}\right) \cdot e\left(g^R, g^{\beta_r}\right) \cdot e\left(g^O, g^{\beta_o}\right) = e\left(g^Z, g\right)$

    which is equivalent to:

    $e(g,g)^{\beta_l L+\beta_r R+\beta_o O} = e(g,g)^Z$

Same variable values tempering technique will fail in such construction because different β-s makes the same polynomials *incompatible* for manipulation. There is however a flaw similar to the one in remark 4.1, concretely because the terms

$$g^{\beta_l}, g^{\beta_r}, g^{\beta_o}$$

are publicly available an adversary can modify the zero-index coefficient of any of the variable polynomials since it does not rely on s, i.e.,

$$g^{\beta_l s^0} = g^{\beta_l}$$

10.16 Non-malleability of Variable and Variable Consistency Polynomials

Let us exemplify remark 4.1 with the following two operations:

$$a \times 1 = b$$

$$3a \times 1 = c$$

The expected result is $b = a$ and $c = 3a$, with clear relationship $c = 3b$. This implies that the *left operand's variable* polynomial has evaluations $l_a(1) = 1$ and $l_a(2) = 3$. Regardless of the form of $l_a(x)$, a prover can unproportionately assign the value of a, by providing modified polynomial $l_a'(x) = al_a(x) + 1$. Therefore evaluations will be $l_a'(1) = a + 1$ and $l_a'(2) = 3a + 1$, hence the results $b = a + 1$ and $c = 3a + 1$ where $c \neq 3b$, effectively meaning that the value of a is different for different operations.
Because the prover has access to

$$g^{\alpha_l} \text{ and } g^{\beta_l}$$

he can satisfy both the *correct operand polynomials* and *variable values consistency* checks:

- ...proving:
  - form left operand polynomial by assigning unproportionately variable a:
    $L(x) = a \cdot l_a(x) + 1$
  - form right operand and output polynomials as usual:
    $R(x) = r_1(x)$, $O(x) = b \cdot o_b(x) + c \cdot o_c(x)$
  - calculate the remainder $h(x) = \frac{L(x) \cdot R(x) - O(x)}{t(x)}$
  - compute encryption: $g^{L(s)} = \left(g^{l_a(s)}\right)^a \cdot g^1$ and as usual for $g^{R(s)}, g^{O(s)}$
  - compute α-shifts: $g^{\alpha L(s)} = \left(g^{\alpha l_a(s)}\right)^a \cdot g^\alpha$ and as usual for $g^{\alpha R(s)}, g^{\alpha O(s)}$
  - compute variable consistency polynomials:
    $g^{Z(s)} = \prod_{i \in \{1,a,b,c\}} \left(g^{\beta_l l_i(s) + \beta_r r_i(s) + \beta_o o_i(s)}\right)^i \cdot g^{\beta_l} = g^{\beta_l(L(s)+1) + \beta_r R(s) + \beta_o O(s)}$

    where the subscript $_i$ represents symbol of the corresponding variable while the exponent ^{i} represents the value of variable; moreover undefined *variable polynomials* are equal to zero.
  - set proof: $\left(g^{L(s)}, g^{R(s)}, g^{O(s)}, g^{\alpha_l L(s)}, g^{\alpha_r R(s)}, g^{\alpha_o O(s)}, g^{Z(s)} g^{h(s)}\right)$
- verification:
  - variable polynomials restriction check:
    $e\left(g^{L'}, g\right) = e\left(g^L, g^\alpha\right) \Rightarrow e\left(g^{\alpha a \cdot l_a(s) + \alpha}, g\right) = e\left(g^{a l_a(s)+1}, g^\alpha\right)$
    and as usually for $g^{R'}, g^{O'}$
  - variable values consistency check
    $e\left(g^L, g^{\beta_l}\right) \cdot e\left(g^R, g^{\beta_r}\right) \cdot e\left(g^O, g^{\beta_o}\right) = e\left(g^Z, g\right) \Rightarrow$
    $e(g,g)^{(a \cdot l_a + 1)\beta_l + R\beta_r + O\beta_o} = e(g,g)^{\beta_l(L+1) + \beta_r R + \beta_o O}$
  - valid operations check $e(g^L, g^R) = e(g^t, g^h) \cdot e(g^O, g)$

10.17 Malleability of Variable Consistency Polynomials

Moreover the availability of:

$$g^{\beta_l}, g^{\beta_r}, g^{\beta_o}$$

allows to use different values of same variable in different operands. For example, if we have an operation:

$$a \quad \times \quad a \quad = \quad b$$

Which can be represented by the variable polynomials:

$$l_a(x) = x, \quad r_a(x) = x, \quad o_a(x) = 0$$
$$l_b(x) = 0, \quad r_b(x) = 0, \quad o_b(x) = x$$

While the expected output is $b = a^2$, we can set different values of a, for example $a = 2$ (left operand), $a = 5$ (right operand) as following:

- proving:
  - ...form left operand polynomial with $a = 2$: $L(x) = 2l_a(x) + 10l_b(x)$
  - form right operand polynomial with $a = 5$: $R(x) = 2r_a(x) + 3 + 10r_b(x)$
  - form output polynomial with $b = 10$: $O(x) = 2o_a(x) + 10o_b(x)$
  - ...compute encryptions:

    $$g^{L(s)} = \left(g^{l_a(s)}\right)^2 \cdot \left(g^{l_b(s)}\right)^{10} = g^{2l_a(s)+10l_b(s)}$$
    $$g^{R(s)} = \left(g^{r_a(s)}\right)^2 \cdot (g)^3 \cdot \left(g^{r_b(s)}\right)^{10} = g^{2r_a(s)+3+10r_b(s)}$$
    $$g^{O(s)} = \left(g^{o_a(s)}\right)^2 \cdot \left(g^{o_b(s)}\right)^{10} = g^{2o_a(s)+10o_b(s)}$$
  - compute variable consistency polynomial:

    $$g^{Z(s)} = \left(g^{\beta_l l_a(s)+\beta_r r_a(s)+\beta_o o_a(s)}\right)^2 \cdot \left(g^{\beta_r}\right)^3 \cdot \left(g^{\beta_l l_b(s)+\beta_r r_b(s)+\beta_o o_b(s)}\right)^{10} =$$
    $$g^{\beta_l(2l_a(s)+10l_b(s)) + \beta_r(2r_a(s)+3+10r_b(s)) + \beta_o(2o_a(s)+10o_b(s))}$$
- verification
  - ...variable values consistency check, should hold:

    $$e\left(g^L, g^{\beta_l}\right) \cdot e\left(g^R, g^{\beta_r}\right) \cdot e\left(g^O, g^{\beta_o}\right) = e\left(g^Z, g\right)$$

Note: polynomials $o_a(x), l_b(x), r_b(x)$ can actually be disregarded since they are evaluating to 0 for any x, however we preserve those for completeness.

Such ability sabotages the *soundness* of proof. It is clear that encrypted β-s should not be available to a prover.

10.18 Non-Malleability

One way to address malleability is to make encrypted β-s from verification key incompatible with encrypted $Z(s)$ by multiplying them in encrypted space by a random secret γ (gamma) during setup stage:

$$g^{\beta_l\gamma},\ g^{\beta_r\gamma},\ g^{\beta_o\gamma}$$

Consecutively such masked encryptions does not allow feasibility to modify encrypted $Z(s)$ in a meaningful way since $Z(s)$ is not a multiple of γ, e.g.,

$$g^{Z(s)} \cdot g^{v' \cdot \beta_l\gamma} = g^{\beta_l(L(s)+\ v'\gamma\ )+\beta_r R(s)+\beta_o O(s)}$$

Because a prover does not know the γ the alteration will be random. The modification requires us to balance the *variable values consistency check* equation in the protocol multiplying $Z(s)$ by γ:

- setup
  - …sample random $\beta_l, \beta_r, \beta_o, \gamma$
  - …set verification key: $\left(\ldots, g^{\beta_l\gamma}, g^{\beta_r\gamma}, g^{\beta_o\gamma}, g^{\gamma}\right)$
- proving …
- verification
  - …variable values consistency check should hold:

$$e\left(g^{L}, g^{\beta_l\gamma}\right) \cdot e\left(g^{R}, g^{\beta_r\gamma}\right) \cdot e\left(g^{O}, g^{\beta_o\gamma}\right) = e\left(g^{Z}, g^{\gamma}\right)$$

It is important to note that we exclude the case when variable polynomials are of 0-degree (e.g., $l_1(x) = 1x^0$), which otherwise would allow to expose encryptions of β in *variable consistency polynomials* of proving key

$$\left\{g^{\beta_l l_i(s)+\beta_r r_i(s)+\beta_o o_i(s)}\right\}_{i\in\{1,\ldots,n\}}$$

in case when any two of operands / output is zero, e.g., for $l_1(x) = 1$, $r_1(s) = 0$, $o_1(s) = 0$ this will result in

$$g^{\beta_l l_1(s)+\beta_r r_1(s)+\beta_o o_1(s)} = g^{\beta_l}$$

We could also similarly *mask* the α-s to address the malleability of *variable polynomials*. However it is not necessary since any modification of a *variable polynomial* needs to be reflected in *variable consistency polynomials* which are not possible to modify.

10.19 Optimization of Variable Values Consistency Check

The *variable values consistency* check is effective now, but it adds 4 expensive pairing operations and 4 new terms to the verification key. The Pinocchio protocol [Par+13] uses a clever selection of the generators g for each operand *ingraining* the "shifts":

- Setup
  - …sample random $\beta, \gamma, \rho_l, \rho_r$ and set $\rho_o = \rho_l \cdot \rho_r$
  - set generators $g_l = g^{\rho_l}, g_r = g^{\rho_r}, g_o = g^{\rho_o}$
  - set proving key:
    $$\left(\left\{g^{s^k}\right\}_{k\in[d]}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}, g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)} \cdot g_r^{\beta r_i(s)} \cdot g_o^{\beta o_i(s)}\right\}_{i\in[n]}\right)$$
  - set verification key: $\left(g_o^{t(s)}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\beta\gamma}, g^{\gamma}\right)$
- Proving
  - …assign variable values
    $$g^{Z(s)} = \prod_{i=1}^{n}\left(g_l^{\beta l_i(s)} \cdot g_r^{\beta r_i(s)} \cdot g_o^{\beta o_i(s)}\right)^{v_i}$$
- Verification
  - …variable polynomials restriction check:
    $e\left(g_l^{L'}, g\right) = e\left(g_l^{L}, g^{\alpha_l}\right)$, and similarly for g_r^R, g_o^O
  - variable values consistency check:
    $e\left(g_l^{L} \cdot g_r^{R} \cdot g_o^{O}, g^{\beta\gamma}\right) = e\left(g^{Z}, g^{\gamma}\right)$
  - valid operations check:
    $e\left(g_l^{L} \cdot g_r^{R}\right) = e\left(g_o^{t}, g^{h}\right) e\left(g_o^{O}, g\right) \Rightarrow$
    $e(g,g)^{\rho_l\rho_r LR} = e(g,g)^{\rho_l\rho_r th + \rho_l\rho_r O}$

Such randomization of the generators further adds to the security making *variable polynomials* malleability, described in remark 4.1, ineffective because for intended change it must be a multiple of either:

$$\rho_l, \rho_r \text{ or } \rho_o$$

raw or encrypted versions of which are not available (assuming, as stated previously that we're not dealing with 0-degree variable polynomials which could expose encrypted versions).
The optimization makes verification key two elements smaller and eliminates two pairing operations from the verification step.
***Note**: there are further protocol improvements in the Jens Groth's 2016 paper [Gro16].*

10.20 Constraints

Our analysis has been primarily focusing on the notion of operation. However, the protocol is not actually "computing" but rather is checking that the output value is the correct result of an operation for the operand's values. That is why it is called a constraint, i.e., a verifier is constraining a prover to provide valid values for the predefined "program" no matter what are they. A multitude of constraints is called a *constraint system* (in our case it is a rank 1 constraint system or R1CS).

Note: This implies that one way to find all correct solutions is to perform a brute-force of all possible combinations of values and select only ones that satisfy the constraints, or use more sophisticated techniques of constraint satisfaction [con18].

Therefore we can also use constraints to ensure other relationships. For example, if we want to make sure that the value of the variable *a* can only be 0 or 1 (i.e., binary), we can do it with the simple constraint:

$$a \times a = a$$

We can also constrain *a* to only be 2:

$$(a-2) \times 1 = 0$$

A more complex example is ensuring that number *a* is a 4-bit number (also called nibble), in other words it is possible to represent *a* with 4 bits. This is quite similar to providing a 'range proof', i.e. a certain value is inside a certain range. We can also call it "ensuring number range" since a 4-bit number can represent 2^4 combinations, therefore 16 numbers in the range from 0 to 15. In the decimal number system any number can be represented as a sum of powers of the base 10 (as the number of fingers on our hands) with corresponding coefficients, for example, $123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$. Similarly a binary number can be represented as a sum of powers of base 2 with corresponding coefficients, for example, 1011 (binary) = $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$ (decimal).

Therefore if *a* is a 4-bit number, then $a = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$ for some boolean b_3, b_2, b_1, b_0. The constraint can be following:

$$1: \quad a \times 1 = 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + 1 \cdot b_0$$

and to ensure that b_3, b_2, b_1, b_0 can only be binary we need to add:

$$2: \quad b_0 \times b_0 = b_0$$

$$3: \quad b_1 \times b_1 = b_1$$

$$4: \quad b_2 \times b_2 = b_2$$

$$5: \quad b_3 \times b_3 = b_3$$

Quite sophisticated constraints can be applied this way, ensuring that the values used are complying with the rules. It is important to note that the above constraint 1 is not possible in the current operation's construction:

$$\sum_{i=1}^{n} c_{l,i} \cdot v_i \times \sum_{i=1}^{n} c_{r,i} \cdot v_i = \sum_{i=1}^{n} c_{o,i} \cdot v_i$$

Because the value 1 (and 2 from the previous constraint) has to be expressed through

$$c \cdot v_{\text{one}}$$

where *c* can be ingrained into the proving key, but the *v* may have any value because the prover supplies it. While we can enforce the $c \cdot v\_one$ to be 0 by setting $c = 0$, it is hard to find a constraint to enforce *v*_one to be 1 in the construction we are limited by. Therefore there should be a way for a verifier to set the value of *v*_one.

10.21 Public Inputs and One

The proofs would have limited usability if it were not possible to check them against the verifier's inputs, e.g., knowing that the prover has multiplied two values without knowing what was the result and/or values. While it is possible to "hardwire" the values to check against (e.g., the result of multiplication must always be 12) in the proving key, this would require to generate separate pair of keys for each desired "verifier's input." Therefore it would be universal if the verifier could specify some of the values (inputs or/and outputs) for the computation, including the *v*_one, instead of the prover.

First, let us consider the proof values

$$g^{L(s)}, g^{R(s)}, g^{O(s)}$$

Because we are using the homomorphic encryption it is possible to augment these values, for example, we can add another encrypted polynomial evaluation

$$g^{L(s)} \cdot g^{l_v(s)} = g^{L(s)+l_v(s)}$$

which means that the verifier could add other variable polynomials to the already provided ones. Therefore if we could exclude necessary variable polynomials from the ones available to the prover, the verifier would be able to set his values on those variables, while the computation check should still match.

It is easy to achieve since the verifier is already constraining the prover in the choice of polynomials he can use empolying the α-shift. Therefore those variable polynomials can be moved from the proving key to the verification key while eliminating its α-s and β checksum counterparts.

The necessary protocol update:

- Setup
  - …separate all n variable polynomials into two groups:
    - verifier's $m+1$:
      $L_v(x) = l_0(x) + l_1(x) + \ldots + l_m$, and alike for $R_v(x)$ and $O_v(x)$,
      where index 0 is reserved for the value of $v_{\text{one}} = 1$
    - prover's $n-m$:
      $L_p(x) = l_{m+1}(x) + \ldots + l_n(x)$, and alike for $R_p(x)$ and $O_p(x)$
  - set proving key:
    $\left(\left\{g^{s^k}\right\}_{k\in[d]}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}, g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)} \cdot g_r^{\beta r_i(s)} \cdot g_o^{\beta o_i(s)}\right\}_{i\in\{m+1,\ldots,n\}}\right)$
  - add to the verification key:
    $\left(\ldots, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i\in\{0,\ldots,m\}}\right)$
- Proving
  - …calculate $h(x)$ accounting for the verifier's polynomials: $h(x) = \frac{L(x)\cdot R(x) - O(x)}{t(x)}$,
    where $L(x) = L_v(x) + L_p(x)$, and similarly for $R(x), O(x)$
  - provide the proof:
    $\left(g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g_l^{\alpha_l L_p(s)}, g_r^{\alpha_r R_p(s)}, g_o^{\alpha_o O_p(s)}, g^{Z(s)}, g^{h(s)}\right)$
- Verification
  - assign verifier's variable polynomial values and add to 1:
    $g_l^{L_v(s)} = g_l^{l_0(s)} \cdot \prod_{i=1}^{m} \left(g_l^{l_i(s)}\right)^{v_i}$
    and similarly for $g_r^{R_v(s)}$ and $g_o^{O_v(s)}$
  - variable polynomials restriction check:
    $e\left(g_l^{L_p}, g^{\alpha_l}\right) = e\left(g_l^{L'_p}, g\right)$ and similarly for $g_r^{R_p}$ and $g_o^{O_p}$
  - variable values consistency check:
    $e\left(g_l^{L_p} g_r^{R_p} g_o^{O_p}, g^{\beta\gamma}\right) = e\left(g^Z, g^\gamma\right)$
  - valid operations check:
    $e\left(g_l^{L_v(s)} g_l^{L_p}, g_r^{R_v(s)} g_r^{R_p}\right) = e\left(g_o^t, g^h\right) \cdot e\left(g_o^{O_v(s)} g_o^{O_p}, g\right)$

Note: following from the protocol properties (Single-Variable Operand Polynomials section) the value 1 represented by polynomials $l_0(x), r_0(x), o_0(x)$ already have appropriate values at the corresponding operations and therefore needs no assignment.

Note: verifier will have to do extra work on the verification step, which is proportionate to the number of variables he assigns.
Effectively this is taking some variables from the prover into the hands of verifier while still preserving the balance of the equation. Therefore the *valid operations* check should still hold, but only if the prover has used the same values that the verifier used for his input.
The value of 1 is essential and allows to derive any number (from the chosen finite field) through multiplication by a constant term, for example, to multiply *a* by 123:

$$1 \cdot a \quad \times \quad 123 \cdot v_{\text{one}} \quad = \quad 1 \cdot r$$

11 Zero-Knowledge Proof of Computation

Since the introduction of the general-purpose computation protocol (Proof of Operation section) we had to let go of the *zero-knowledge* property, to make the transition simpler. Until this point, we have constructed a verifiable computation protocol.
Previously to make a proof of polynomial *zero-knowledge* we have used the random δ-shift, which makes the proof indistinguishable from random (Zero-Knowldge section):

$$\delta p(s) = t(s) \cdot \delta h(s)$$

With the computation we are proving instead that:

$$L(s) \cdot R(s) - O(s) = t(s)h(s)$$

While we could just adapt this approach to the multiple polynomials using same δ, i.e., supplying randomized values $\delta L(s)$, $\delta R(s)$, $\delta^2 O(s)$, $\delta^2 h(s)$, which would satisfy the *valid operations check* through pairings:

$$e(g,g)^{\delta^2 L(s)R(s)} = e(g,g)^{\delta^2(t(s)h(s)+O(s))}$$

The issue is that having same δ hinders security, because we provide those values separately in the proof:

- one could easily identify if two different polynomial evaluations have same value, learning some knowledge, e.g.:
$$g^{\delta L(s)} = g^{\delta R(s)}$$
- potential insignificance of differences of values between $L(s)$ and $R(s)$ could allow factoring of those differences through brute-force, for example if $L(s) = 5R(s)$, iterating check
$$g^{L(s)} = \left(g^{R(s)}\right)^i$$

for $i \in \{1 \ldots N\}$ would reveal the 5× difference in just 5 steps. Same brute-force can be performed on encrypted addition operation, e.g.,

$$g^{L(s)} = g^{R(s)+5}$$

- other correlations between elements of the proof may be discovered, for example, if
$$e(g^{\delta L(s)}, g^{\delta R(s)}) = e(g^{\delta^2 O(s)}, g)$$

then $L(x) \cdot R(x) = O(x)$, etc.

Note: the consistency check optimization makes such data mining harder but still allows to discover relationships, apart from the fact that verifier can choose ρ_l, ρ_r in a particular way that can facilitate revealing of knowledge (as long as it is not a diversified setup).
Consequently, we need to have different randomness (δ-s) for each polynomial evaluation, e.g.:

$$\delta_l L(s) \cdot \delta_r R(s) - \delta_o O(s) = t(s) \cdot (\Delta \textcircled{?} h(s))$$

To resolve inequality on the right side, we can only modify the proof's value $h(s)$, without alteration of the protocol which would be preferable. Delta (Δ) here represents the difference we need to apply to $h(s)$ in order to counterbalance the randomness on the other side of the equation and ?◯represents either multiplication or addition operation (which in turn accommodates division and subtraction). If we chose to apply Δ through multiplication (?◯ = ×) this would mean that it is impossible to find Δ with overwhelming probability, because of randomization:

$$\Delta = \frac{\delta_l L(s) \cdot \delta_r R(s) - \delta_o O(s)}{t(s)h(s)}$$

We could set:

$$\delta_o = \delta_l \cdot \delta_r$$

which transforms into:

$$\Delta = \frac{\delta_l \delta_r (L(s) \cdot R(s) - O(s))}{t(s)h(s)} = \delta_l \delta_r$$

However, as noted previously this hinders the zero-knowledge property, and even more importantly such construction will not accommodate the verifier's input polynomials since they must be multiples of the corresponding δ-s, which would require an interaction. We can try adding randomness to the evaluations:

$$(L(s) + \delta_l) \cdot (R(s) + \delta_r) - (O(s) + \delta_o) = t(s) \cdot (\Delta \times h(s))$$

$$\Delta = \frac{\overbrace{L(s)R(s) - O(s)}^{t(s)h(s)} + \delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_o}{t(s)h(s)} = 1 + \frac{\delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_o}{t(s)h(s)}$$

However due to randomness it is non-divisible. Even if we address this by multiplying each δ with $t(s)h(s)$, because we apply Δ through multiplication of $h(s)$, and Δ will consist of encrypted evaluations (i.e., $E(L(s))$, etc.) it will not be possible to compute

$$g^{\Delta h(s)}$$

without use of pairings (result of which is in another number space). Likewise computation is not possible through encrypted evaluation of $\Delta h(x)$ using encrypted powers of s (from 1 to d), because the degree of $h(x)$ and Δ is d, hence the degree of $\Delta h(x)$ is up to $2d$. Moreover, it is not possible to compute such randomized operand polynomial evaluation for the same reason:

$$g^{L(s) + \delta_l t(s)h(s)}$$

Therefore we should try applying Δ through addition (?◯ = +), since it is available for homomorphically encrypted values.

$$(L(s)+\delta_l)\cdot(R(s)+\delta_r)-(O(s)+\delta_o)=t(s)\cdot(\Delta+h(s))$$

$$\Delta=\frac{L(s)R(s)-O(s)+\delta_r L(s)+\delta_l R(s)+\delta_l\delta_r-\delta_o-t(s)h(s)}{t(s)}\Rightarrow$$

$$\Delta=\frac{\delta_r L(s)+\delta_l R(s)+\delta_l\delta_r-\delta_o}{t(s)}$$

Every term in the numerator is a multiple of a δ, therefore we can make it divisible by multiplying each δ with $t(s)$:

$$(L(s)+\delta_l t(s))\cdot(R(s)+\delta_r t(s))-(O(s)+\delta_o t(s))=t(s)\cdot(\Delta+h(s))$$

$$\cancel{L(s)R(s)-O(s)}+t(s)(\delta_r L(s)+\delta_l R(s)+\delta_l\delta_r t(s)-\delta_o)=t(s)\Delta+\cancel{t(s)h(s)}$$

$$\Delta=\delta_r L(s)+\delta_l R(s)+\delta_l\delta_r t(s)-\delta_o$$

Which we can efficiently compute in the encrypted space:

$$g^{L(s)+\delta_l t(s)}=g^{L(s)}\cdot\left(g^{t(s)}\right)^{\delta_l},\quad \text{etc.}$$

$$g^{\Delta}=\left(g^{L(s)}\right)^{\delta_r}\cdot\left(g^{R(s)}\right)^{\delta_l}\cdot\left(g^{t(s)}\right)^{\delta_l\delta_r}g^{-\delta_o}$$

This leads to passing of *valid operations* check while concealing the encrypted values.

$$L\cdot R-O+t(\delta_r L+\delta_l R+\delta_l\delta_r t-\delta_o)=t(s)h+t(s)(\delta_r L+\delta_l R+\delta_l\delta_r t-\delta_o)$$

The construction is statistically *zero-knowledge* due to addition of uniformly random multiples of δ_l, δ_r, δ_o (see theorem 13 of [Gen+12]).
Note: this approach is also consistent with the verifier's operands, e.g.:

$$g_l^{L_p+\delta_l t}\cdot g_l^{L_v}=g_l^{L_p+L_v+\delta_l t}$$

therefore the valid operations check holds but still only if the prover have used verifier's values to construct the proof (i.e., $\Delta=\delta_r(L_p+L_v)+\delta_l(R_p+R_v)+\delta_l\delta_r t-\delta_o$), see next section for more details.
To make the "variable polynomials restriction" and "variable values consistency" checks coherent with the *zero-knowledge* alterations, it is necessary to add the following parameters to the proving key:

$$g_l^{t(s)},g_r^{t(s)},g_o^{t(s)},g_l^{\alpha_l t(s)},g_r^{\alpha_r t(s)},g_o^{\alpha_o t(s)},g_l^{\beta t(s)},g_r^{\beta t(s)},g_o^{\beta t(s)}$$

It is quite curious that the original Pinocchio protocol [Par+13] was concerned primarily with the verifiable computation and less with the *zero-knowledge* property, which is a minor modification and comes almost *for free*.

12 zk-SNARK Protocol

Considering all the gradual improvements the final zero-knowledge succinct non-interactive arguments of knowledge, aka Pinocchio [Par+13], protocol is (the *zero-knowledge* components are optional and highlighted with a *different color*):

- Setup
  - select a generator g and a cryptographic pairing e
  - for a function $f(u) = y$ with n total variables of which m are input/output variables, convert into the polynomial form (quadratic arithmetic program) $\left(\{l_i(x), r_i(x), o_i(x)\}_{i\in\{0,\dots,n\}}, t(x)\right)$ of degree d (equal to the number of operations) and size $n+1$
  - sample random $s, \rho_l, \rho_r, \alpha_l, \alpha_r, \alpha_o, \beta, \gamma$
  - set $\rho_o = \rho_l \cdot \rho_r$ and the operand generators $g_l = g^{\rho_l}, g_r = g^{\rho_r}, g_o = g^{\rho_o}$
  - set the proving key:
    $\Big(\left\{g^{s^k}\right\}_{k\in[d]}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i\in\{0,\dots,n\}},$
    $\left\{g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)} g_r^{\beta r_i(s)} g_o^{\beta o_i(s)}\right\}_{i\in\{m+1,\dots,n\}},$
    $g_l^{t(s)}, g_r^{t(s)}, g_o^{t(s)}, g_l^{\alpha_l t(s)}, g_r^{\alpha_r t(s)}, g_o^{\alpha_o t(s)}, g_l^{\beta t(s)}, g_r^{\beta t(s)}, g_o^{\beta t(s)} \Big)$
  - set the verification key:
    $\left(g^1, g_o^{t(s)}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i\in\{0,\dots,m\}}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\gamma}, g^{\beta\gamma}\right)$
- Proving
  - for the input u, execute the computation of $f(u)$ obtaining values $\{v_i\}_{i\in\{m+1,\dots,n\}}$ for all the itermediary variables
  - assign all values to the unencrypted variable polynomials $L(x) = l_0(x) + \sum_{i=1}^{n} v_i \cdot l_i(x)$ and similarly $R(x), O(x)$
  - sample random δ_l, δ_r and δ_o
  - find $h(x) = \dfrac{L(x)R(x) - O(x)}{t(x)} + \delta_r L(x) + \delta_l R(x) + \delta_l \delta_r t(x) - \delta_o$
  - assign the prover's variable values to the encrypted variable polynomials and apply *zero-knowledge* δ-shift $g_l^{L_p(s)} = \left(g_l^{t(s)}\right)^{\delta_l} \cdot \prod_{i=m+1}^{n} \left(g_l^{l_i(s)}\right)^{v_i}$
    and similarly $g_r^{R_p(s)}$, $g_o^{O_p(s)}$
  - assign its α-shifted pairs $g_l^{L'_p(s)} = \left(g_l^{\alpha_l t(s)}\right)^{\delta_l} \cdot \prod_{i=m+1}^{n} \left(g_l^{\alpha_l l_i(s)}\right)^{v_i}$
    and similarly $g_r^{R'_p(s)}$, $g_o^{O'_p(s)}$
  - assign the variable values consistency polynomials
    $g^{Z(s)} = \left(g_l^{\beta t(s)}\right)^{\delta_l} \left(g_r^{\beta t(s)}\right)^{\delta_r} \left(g_o^{\beta t(s)}\right)^{\delta_o} \cdot \prod_{i=m+1}^{n} \left(g_l^{\beta l_i(s)} g_r^{\beta r_i(s)} g_o^{\beta o_i(s)}\right)^{v_i}$
  - compute the proof $\left(g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g^{h(s)}, g_l^{L'_p(s)}, g_r^{R'_p(s)}, g_o^{O'_p(s)}, g^{Z(s)}\right)$
- Verification
  - parse a provided proof as $\left(g_l^{L_p}, g_r^{R_p}, g_o^{O_p}, g^{h}, g_l^{L'_p}, g_r^{R'_p}, g_o^{O'_p}, g^{Z}\right)$
  - assign input/output values to verifier's encrypted polynomials and add to 1:
    $g_l^{L_v(s)} = g_l^{l_0(s)} \cdot \prod_{i=1}^{m} \left(g_l^{l_i(s)}\right)^{v_i}$ and similarly for $g_r^{R_v(s)}$ and $g_o^{O_v(s)}$
  - variable polynomials restriction check :
    $e\left(g_l^{L_p}, g^{\alpha_l}\right) = e\left(g_l^{L'_p}, g\right)$ and similarly for $g_r^{R_p}$ and $g_o^{O_p}$
  - variable values consistency check:
    $e\left(g_l^{L_p} g_r^{R_p} g_o^{O_p}, g^{\beta\gamma}\right) = e\left(g^{Z}, g^{\gamma}\right)$
  - valid operations check:
    $e\left(g_l^{L_p} g_l^{L_v(s)}, g_r^{R_p} g_r^{R_v(s)}\right) = e\left(g_o^{t(s)}, g^{h}\right) \cdot e\left(g_o^{O_p} g_o^{O_v(s)}, g\right)$

13 Conclusions

We ended up with an effective protocol which allows proving computation:

- succinctly — independently from the amount of computation the proof is of constant, small size
- non-interactively — as soon as the proof is computed it can be used to convince any number of verifiers without direct interaction with the prover
- with argumented knowledge — the statement is correct with non-negligible probability, i.e., fake proofs are infeasible to construct; moreover prover knows the corresponding values for the true statement (i.e. witness), e.g., if the statement is "B is a result of sha256(a)" then the prover knows some A such that B = sha256(a) which is useful since B could only be computed with the knowledge of a as well as it's infeasible to compute a from B (assuming a have enough entropy)
- the statement is correct with non-negligible probability, i.e., fake proofs are infeasible to construct
- in *zero-knowledge* — it is "hard" to extract any knowledge from the proof, i.e., it is indistinguishable from random

It was possible to achieve primary due to unique properties of polynomials, modular arithmetic, homomorphic encryption, elliptic curve cryptography, cryptographic pairings and ingenuity of the inventors.
This protocol proves computation of a unique finite execution machine which in one operation can add together almost any number of variables but may only perform one multiplication. Therefore there is an opportunity to both optimize programs to leverage this specificity efficiently as well as use constructions which minimize the number of operations.
It is essential that verifier does not have to know any secret data in order to verify a proof so that properly constructed verification key can be published and used by anyone in a non-interactive manner. Which is contrary to the "designated verifier" schemes where the proof will convince only one party, therefore it is non-transferable. In *zk-SNARK* context, we can achieve this property if untrustworthy or a single party generates the keypair.
The field of zero-knowledge proof constructions is continuously evolving, introducing optimizations ([BCTV13, Gro16, GM17]), improvements such as updatable proving and verification keys ([Gro+18]), and new constructions (Bulletproofs [Bün+17], ZK-STARK [Ben+18], Sonic [Mal+19]).