You Could Have Invented SNARKs

Ethan Buchman

December 14, 2024

zk-SNARKs are beautiful constructions shrouded in mystery. They open new horizons in private and scalable compute. They are essential to privacy-preserving digital transactions and to the scalability roadmap of all blockchains.

But the basics can actually be relatively easily understood by someone with no more than high school math (basic familiarity with polynomials and exponent rules). Perhaps you could have even invented them yourself, if you had tried. So let's try.

Here we'll introduce a zk-SNARK without making any leaps, where you can follow every step, and where every thing we do is clearly motivated and almost obvious so that in principle you could have thought of it yourself.

We're going to do this with the first practical zk-SNARK, which was invented in 2013, and is called Pinocchio. Pinocchio has all the pieces necessary for publicly verifiable computation, and is the SNARK used in the first version of ZCash. Pinocchio was improved a few years later by Groth with his famous "Groth16", which is now the most popular and widely deployed SNARK. Groth16 is effectively an optimization of Pinocchio, and its easier to understand Pinocchio first. Since Groth16, and especially since 2019 (Sonic, Plonk, snarktember, etc.), there has been an explosion of new SNARKs and perhaps more importantly, new ways to understand and explain them. But I think it's easiest and most accessible to start from trying to invent Pinocchio. Once you understand it, you can generalize from there.

So what's a zk-SNARK?. It stands for Zero-Knowledge Succinct Non-interactive ARgument of Knowledge. A mouthful. At it's core it's just a proof that a specific computation was done correctly ("ARgument of Knowledge"), that is verifiable by anyone ("Non-interactive"), that is short and cheap to verify ("Succinct"), and that doesn't reveal anything about the inputs to the computation ("Zero-Knowledge).

Lots to unpack, but we'll start simple. Let's just see if we can get one person to prove to another that a specific computation was done correctly, without them having to redo it. We could call this a Succinct ARgument of Knowledge, or SARK. Then we'll see if we can make it so anyone can verify the proof, making it Non-interactive, or a SNARK. Then we'll see if we can hide the inputs, making it Zero-Knowledge, a zk-SNARK. Finally, we'll correct for any differences in the actual Pinocchio paper.

Let's do it.

Contents

1 Setup - A simple example **3**

2 Intuition for a Proof **5**

3 Maybe Polynomials? **6**

3.1 Encoding Many Values 6
3.2 Division 7
3.3 Secret Points 8

4 Encoding Computation in Polynomials **9**

4.1 Encoding the Gates 10
4.2 Encoding the Wires 11
4.3 Encoding Individual Wires 13

5 Returning to a Proof **16**

5.1 Math on Encrypted Data 17
5.2 Proving Knowledge of a Secret Exponent 19

6 Breaking up the Proof **21**

6.1 Proving a Linear Combination 21
6.2 Proving *the same* linear combination 23

7 Completing the Proof **25**

7.1 Evaluating an Encrypted Polynomial 26
7.2 Encrypted Multiplication 26
7.3 Separating input/output values from intermediate values 28

8 Summarizing the Proof: a SARK **29**

8.1 Setup 32
8.2 Prove 33
8.3 Verify 34

9 Shared Setup and a Non-Interactive Proof: the SNARK **36**

9.1 Setup 38
9.2 Prove 39
9.3 Verify 39

10 Completing Pinocchio: Constants, Addition, Zero Knowledge, and Optimizations 41
10.1 Addition and Constants 41
10.2 Zero Knowledge 44
10.2.1 Setup 48
10.2.2 Prove 49
10.2.3 Verify 50
10.3 Optimization 50
10.3.1 Setup 52
10.3.2 Prove 53
10.3.3 Verify 53

11 The World of Modern SNARKs 54

12 Acknowledgments 56

1 Setup - A simple example

Suppose Veronica wants to do a computation, like multiplying a number by itself a few hundred times. But Veronica is bad at doing lots of multiplication, and she wants to outsource the work to her friend, Philip. Veronica doesn't fully trust Philip, so she wants him to also prove he did the multiplication correctly. Of course, this only makes sense if it's easier for Veronica to verify the proof with less work than it took to do the original multiplication. In cryptography language we say Philip is the Prover and Veronica is the Verifier. But we'll stick with the human names.

So let's think about how we might do this, using an example. Suppose Veronica wanted to multiply some number x by itself five times. In other words, she wants to compute the value $y = x^5$. We can compute x^5 one multiplication at a time, multiplying by x over and over: $x^5 = (x * x) * x) * x) * x$. This computation has an input (x), three intermediate values, and a final output.[1] Since we need Philip to prove he did this computation, we probably need him to keep track of all these intermediate values, so he can prove he knows them. Proving you did a computation correctly really just boils down to proving you *know* all the intermediate values between the input and output values.

Lets start by labeling all these values. We can refer to them as C_i for the 5 values of i, with $C_1 = x$. Then each C_i is just x^i, and the final output is $C_5 = x^5$:

$$C_1 = x; C_2 = x^2; C_3 = x^3; C_4 = x^4; C_5 = x^5$$

[1]Maybe you realized we can do this with fewer multiplications by squaring x first: $x^5 = x * x^2 * x^2$. We'll leave that aside for now.

When we just write $y = x^5$ we only see the input x, the output y, and the operations applied to x. But we don't see any of the intermediate values.

To see them better, we can try to represent the computation visually, using the C_i. We want to visually show how inputs are combined by operations into intermediate values that are further combined with inputs or other intermediate values successively into output values. We could lay it out as a graph where values (input/intermediate/output) are edges and operations are nodes:

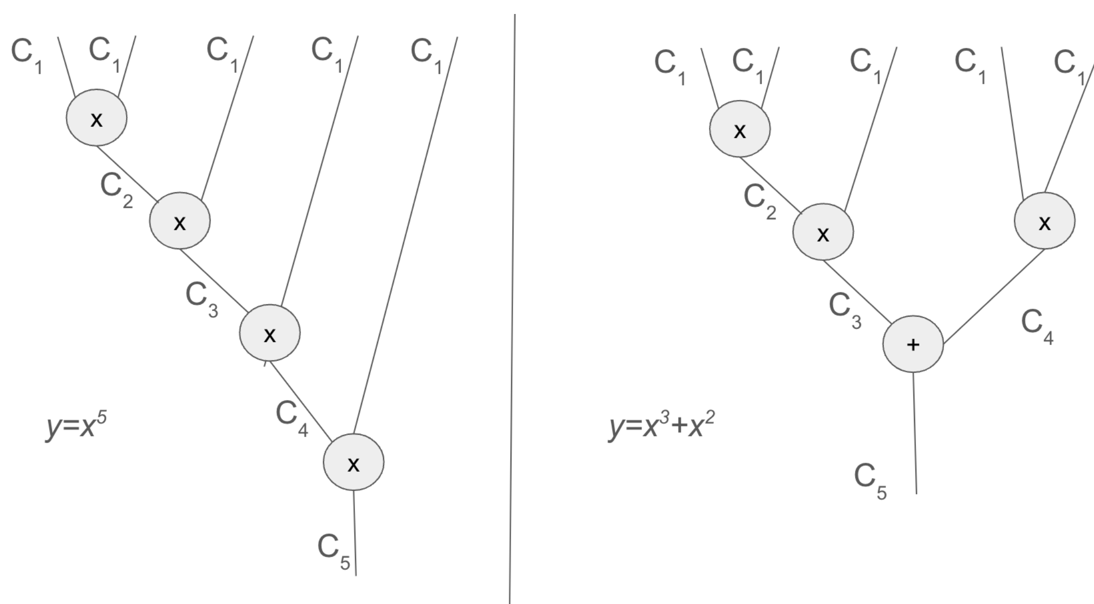

Figure 1: On the left, a graphical representation of $y = x^5$ that shows all the intermediate values. On the right, one for $y = x^3 + x^2$

This idea of laying out a function or computation like $y = x^5$ graphically is well known in computer science and especially in hardware and electrical engineering. In that world, these graphs are called "circuits" because they lay out the full computation as a physical circuit of "wires" (the edges) passing through "gates" (the nodes).

It turns out, any computation can be represented in this way as a circuit, using wires for the values and gates for the operations. For now, we're just thinking about multiplication. Notice how each multiplication gate has two input wires and one output wire. Every wire is labeled with its value.

So this is the problem Philip is faced with. He needs to compute the value C_5 (by first computing the intermediate values C_2, C_3, and C_4), and then he needs to prove to Veronica that he did the computation correctly. Most importantly, Veronica doesn't want to have to know about the values of C_2, C_3, and C_4. She knows they exist, of course, but when she provides the number C_1 to be multiplied 5 times, she doesn't care about the intermediate values. She only cares about the result and that it's correct. Looking at our circuit diagram, both Philip and Veronica know the computation that needs to be

performed (the structure of the gates and how they're connected), but only Philip will know the value of all the intermediate wires.

How can Philip prove to Veronica he did the multiplications correctly? In other words, how can he prove that he knows the values of all the intermediate wires and they're all correct?

2 Intuition for a Proof

One way cryptographers try to approach problems of proving things is by relying on functions that are easy to verify but hard to actually compute. The most basic example of this is that it's easier to multiply than to divide. We're actually taught to check if we did division correctly by multiplying. The same goes for taking logarithms, which is hard, while exponents are easy. Cryptographers literally call these kinds of division/logarithm problems "hard", and they operate with kinds of numbers (really big ones) where doing division is so hard it could take all the world's supercomputers thousands of years, while the multiplication is so easy anyone can do it on a phone. Practically all cryptography depends on these kinds of "hardness assumptions."

How does that help us? In our case, Philip knows all the wires, and Veronica only knows the inputs and gates. Lets call what Philip knows W and what Veronica knows G. Is there some way we can structure W and G so that Philip could only divide them if he really did everything correctly?

Suppose for a second W and G are just numbers that somehow represent all the wires and all the gates. If Philip can divide W by G, it means W is a multiple of G, so there must be some H such that $W/G = H$ and thus $W = G * H$. Since Veronica knows G (she knows the gates), Philip could then just give her W and H and she could check for herself that indeed $W = G * H$. But does this really work? Philip could just pick a random H and then multiply it with G to get a W. How would Veronica ever know that W was actually computed from the actual values of the wires?

And further, how can we even encode what Philip knows, or what Veronica knows, as a single number W or G? For Philip, he knows a bunch of numbers (all the inputs, intermediate values, and outputs). If we try to combine all these numbers into one single number, we will likely lose a lot of structure and information. We need some other way to encode a set of numbers into a single mathematical object that we can still multiply and divide.

Keep in mind we're trying to figure out a scheme here. There's no obvious way to do what we want to do, we're going to have to put some different pieces together and try some different things to see if they work. What we've currently got is the idea that perhaps we can encode the wires into a single mathematical object W and the gates into an object G such that W will be a multiple of G *if and only if* the value of the wires

encoded in W were correctly computed according to the gates encoded in G.

So what are these mathematical objects W and G that can encode the wires and gates in such a way that preserves all the information but still allows W to be divided by G?

3 Maybe Polynomials?

The main kind of mathematical object we're taught in high school that combines multiple numbers is a polynomial. We also learn that polynomials can be added, subtracted, multiplied, and divided. That's convenient, because often when mathematicians and cryptographers want to encode a bunch of numbers into a single mathematical object, they also reach for polynomials. Why? A few reasons.

3.1 Encoding Many Values

First is that polynomials are inherently defined in terms of a set of numbers. Take the familiar quadratic $y = ax^2 + bx + c$. This is effectively an encoding of the three numbers a, b, c, the polynomials so-called "coefficients." So any polynomial of degree n can be understood as an encoding of the $n + 1$ numbers that define its coefficients.

But there's another way polynomials can encode numbers. And cryptographers especially love this. Any set of n points in a Cartesian plane defines a unique polynomial of degree $n - 1$. You can see this most simply in the case of a line. Any set of two points in the plane uniquely defines a line passing through those two points (recall a line has degree 1). But once you write down the equation of the line, there's an infinite number of points that fall on that line. So the line encodes the original two points, but it also hides them, because given the line, you don't know what the original two points were!

Take a simple example. Suppose I pick the points $(1, 5)$ and $(2, 7)$. I just randomly picked these. They uniquely define the line $y = 2x + 3$ (there's no other straight line that passes through both these points). But if all you knew was the line $y = 2x + 3$, there's no way you could know that I started with the points $(1, 5)$ and $(2, 7)$, since there's a zillion other points on the line - points like $(0, 3)$, $(-10, 17)$, $(100, 203)$, and so on.

The same goes for any set of n points. If I selected 3 points, they would uniquely define a quadratic function (degree 2). 4 points would uniquely define a cubic (degree 3). And so on. In this way we can define a unique polynomial from any set of points.

There's one more way for polynomials to encode a set of numbers, though in this case it's not necessarily unique, and that's by setting the roots of the polynomial to those numbers. In other words, if we have the numbers r, s, t we can define a polynomial $y = (x - r)(x - s)(x - t)$. Of course we can multiply this by any constant k, so there are many polynomials of degree n with the same n roots. But its still potentially a useful encoding.

Summary: How to Encode Numbers in a Polynomial

To summarize, there are three ways to encode a set of numbers as a polynomial:

1. Coefficients: any n values defines a polynomial of degree $n-1$ by using them as coefficients in the standard form. e.g. the 3 values (a, b, c) in $y = ax^2+bx+c$
2. Points: any n points uniquely define a polynomial of degree $n-1$ as the polynomial that passes through those points
3. Zeros: any n values defines a family of polynomials of degree n by using them as the roots or factored form of the polynomial, e.g the 3 values (a, b, c) in $y = k(x-a)(x-b)(x-c)$

The third way might be especially useful because it gives us a natural way to represent one polynomial as a multiple of another. But we will actually encounter each of these as we invent SNARKs.

3.2 Division

So polynomials might be a useful way to encode numbers and points. That's nice. But polynomials also have some other useful properties, namely, they can behave like numbers. You can add and subtract and multiply them. But you can actually even divide them. Suppose we had a polynomial $W(x)$ and another polynomial $G(x)$. Just like with numbers, saying $W(x)$ can be divided by $G(x)$ is the same as saying $W(x)$ is a multiple of $G(x)$. And just like with numbers, if $W(x)$ is a multiple of $G(x)$ then there must be some other polynomial $H(x)$ such that $W(x) = G(x) * H(x)$.

You may recall from high school math that just like numbers can be factored into their 'prime' number factors (e.g. $42 = 2 * 3 * 7$), polynomials can be factored into their 'root' polynomials. For instance the polynomial $W(x) = x^2 + x - 6$ can be factored as $W(x) = (x+3)(x-2)$. If we said $G(x) = (x+3)$ then we can see that $G(x)$ divides $W(x)$ (i.e. W is a multiple of G) and that $H(x) = (x-2)$ such that $W(x) = G(x) * H(x)$.

This was a bit of a detour, but recall we're looking for a mathematical object we can use to encode our wires and gates. Between their ability to encode sets of numbers and their ability to be divided, polynomials seem promising! We would encode the wires as a polynomial $W(x)$ and the gates as a polynomial $G(x)$ such that $W(x)$ is a multiple of $G(x)$ but only if all the values of the wires encoded in W were correctly computed based on the gates encoded in G (how exactly to do this encoding we still have to figure out). Recall also that Veronica knows the gates, and so can compute $G(x)$ for herself, but only Philip knows all the wires. So Philip will have to send $W(x)$ and $H(x)$ to Veronica and she will have to check that $W(x) = G(x) * H(x)$.

This raises two points. First, as we already saw, Philip might have made up H and computed W from H and G and Veronica has no way to know that W was created properly (as an encoding of the values of the wires). But since polynomials have more structure than individual numbers, there might be ways for Veronica to check the polynomial $W(x)$ was created correctly. Let's hope we can figure that out.

Second, If Philip sends Veronica the polynomials $W(x)$ and $H(x)$ so she can check $W(x) = G(x) * H(x)$, then she will have to multiply $G(x)$ and $H(x)$, which could itself require multiplying many numbers, possibly as many or more than the original computation we're trying to verify. That would completely defeat the purpose of outsourcing the multiplication in the first place! We need to ensure that the work Veronica does is much less than the work Philip does, especially as the complexity of the computation increases. In our example we're only multiplying a number by itself 5 times, but imagine we were multiplying a number by itself 1000 times, or whatever.

3.3 Secret Points

Is there some way Veronica can check $W(x) = G(x) * H(x)$ more cheaply? For instance, what if she picked a special random value, call it s, and sent it to Philip. Then instead of Philip sending whole polynomials $W(x)$ and $H(x)$ he could just send the two numbers $W(s)$ and $H(s)$. Veronica could then just check the one multiplication $W(s) = G(s) * H(s)$, and avoid having to multiply whole polynomials.

But how do we know this will work? If $W(s) = G(s) * H(s)$ for a single value s chosen by Veronica, how do we know that $W(x) = G(x) * H(x)$ for all values of x? Veronica could pick a bunch of values for s, and check for all of them. But how many is enough? Earlier we saw that a polynomial of degree n is uniquely defined by $n + 1$ points (i.e. lines are defined by 2 points, quadratics by 3, cubics by 4, etc.). If you have two different polynomials of degree n, they are only going to intersect each other at most at n points (i.e. lines only intersect at one point, quadratics at 2, etc.). So if we check $n + 1$ points and they're all equal, we should be guaranteed the polynomials are the same. But what if our polynomials have really high degree? We don't want Veronica checking thousands or millions of points.

Lets think about her just randomly picking one point again. If we have two polynomials of degree n, we know they might intersect at $n+1$ points, but if Veronica randomly picks one of the zillions of possible points on those polynomials, what are the chances it will equal one of the n points where they intersect?

Good question. And your intuition is correct. In fact you just came up with a famous result called the "Schwartz–Zippel lemma" which says just that:

Schwartz-Zippel Lemma

so long as the degree of the polynomial is small enough, then if you have two polynomials $f(x)$ and $g(x)$, and you randomly pick a large value s, and $f(s) = g(s)$, then there is extremely high probability that the polynomials are the same (i.e. $f(x) = g(x)$).

This is kind of insane since it means that you can check if two polynomials are the same everywhere just by checking if they're the same at one single point! Wild.

But what does it mean for the degree of the polynomial to be "small enough"? Well, every polynomial is defined over what mathematicians call a "field". The field is just the set of all possible numbers that the coefficients of the polynomial and the points on the polynomial can come from. In high school the field is the set of all real numbers. But we can also define polynomials over other fields like the integers, or complex numbers, or even over so-called 'finite fields' like those defined by elliptic curves (we'll come back to that). All that matters for Schwartz-Zippel is that the degree of the polynomial is significantly less than the size of the field. Since we're usually dealing with very large fields (in the case of real numbers, its infinite size, in the case of finite fields defined by elliptic curves, the size is massive, like 2^{256}), then even polynomials with degrees in the millions are small enough for Schwartz-Zippel to apply.

This should sound pretty wild, if not unbelievable. What we've effectively said is that we can encode millions of values into a polynomial of degree in the millions (like, $x^{1000000}$). Do division on it. And then check if the division was correct by doing a single multiplication. Seems like voodoo, and maybe should scare you a bit.

But if you think about it, it's not really that different from what makes your Bitcoin private key secure. There's like 2^{256} private keys out there. This number is so big, that even if millions or billions of people randomly generate millions of keys each the chances of any of them generating the same key is basically 0. So we don't worry about it. Same goes here.

Awesome! That means if we can figure out how to encode our wires and gates correctly as polynomials $W(x)$ and $G(x)$, Veronica can verify the computation by checking the polynomials at a single point s, i.e. $W(s) = G(s) * H(s)$. But how on earth are we going to encode wires and gates into a polynomial?!

4 Encoding Computation in Polynomials

We need to figure out a way to encode the wires (values from the computation) into a polynomial $W(x)$. And we need to encode the gates into a polynomial $G(x)$. And we need it to be the case that $W(x)$ is a multiple of $G(x)$ *if and only if* the values of the

wires encoded into $W(x)$ were correctly computed according to the gates encoded into $G(x)$. This is going to be a bit tricky.

We saw in the last section that there are three ways to encode a bunch of numbers into a polynomial. The third way we presented, which encoded the numbers as the roots or zeros of the polynomial, lent itself most naturally to doing multiplication and division. If we have some numbers a, b, c, d we encode them as the roots of a polynomial $y = (x - a)(x - b)(x - c)(x - d)$. Note this means the polynomial is actually 0 at all of those values. We want $W(x)$ to be a multiple of $G(x)$, so let's start with $G(x)$.

4.1 Encoding the Gates

One problem here is that while we have values for the wires (C_1, C_2, ..., C_5), we don't actually have values for the gates. The gates are just multiplications, there's no actual numbers associated with them. The structure of the gates is defined by the wires that connect them. So what does it even mean to encode them into a polynomial?

At the very least, we can encode how many gates there are. And since we'll need some way for $W(x)$ to refer to the different gates (it needs to encode how the different wires connect through the different gates), it would probably help to number or label each gate. The simplest thing to do is just number them in order. In our case, since we have 4 gates, we can number them 1, 2, 3, and 4.

We now want to encode these gates as a polynomial $G(x)$. But we also know we're going to want $W(x)$ to be a multiple of $G(x)$. We already saw how to encode a set of numbers as a polynomial in such a way that its easy to make multiples of that polynomial - just use those numbers as the roots of the polynomial. So let's encode these gate numbers by having them be the roots.

Encoding the Gates

In other words, $G(x) = (x - 1)(x - 2)(x - 3)(x - 4)$, with one root for each gate, with gates numbered 1, 2, 3, ...

This does feel a bit weird. We just made up those numbers, and already we're encoding them in a polynomial. We could have used whatever numbers we wanted, but might as well number the gates simply and in order. At this point $G(x)$ just tells us that there's 4 gates and they're numbered 1 to 4. As a polynomial, x could be any value, but we're choosing 4 values of x to be special values that correspond to the existence of gates. At these values of x, the polynomials $G(x)$ and $W(x)$ are zero.

4.2 Encoding the Wires

Now we have to define $W(x)$. Again, we need $W(x)$ to be a multiple of $G(x)$ (so that Veronica can later easily check). And we need $W(x)$ to correctly encode the values of all the wires, which must be computed correctly according to the multiplication gates. This is the tricky part. We know that C_1 is given, but it must be the case that $C_2 = C_1 * C_1$ and $C_3 = C_2 * C_1$ and $C_4 = C_3 * C_1$ and $C_5 = C_4 * C_1$, corresponding to our four multiplication gates. How can we encode all this?

Well, recall that we assigned values to our four multiplication gates: 1, 2, 3, and 4. And by definition, $G(x) = 0$ when x equals any of these 4 values. If $W(x)$ is going to be a multiple of $G(x)$, it will also have to equal 0 at these four values. It might be 0 at more values, but at least it has to be zero at these.

So what do? Notice that for each gate, we have a left input, a right input, and an output. And of course, the left input times the right input equals the output:

$$C_1 * C_1 = C_2$$

Or, in other words, the left input times the right input minus the output equals 0

$$C_1 * C_1 - C_2 = 0$$

Let's try to start as simple as possible. What if we only had one gate, call it gate 1. And we're just trying to compute $C_2 = C_1 * C_1$. We already came up with an idea for how to encode $G(x)$, so in this case since we labeled it gate 1 it would just be $G(x) = (x - 1)$. And now we want to encode the wires into a polynomial $W(x)$ that's a multiple of $G(x)$.

Remember these are polynomials, so they're defined at any value of x. We're trying to define these polynomials by constraining them to be 0 when x equals the value of a gate. In our simple example with one gate, we have $G(x) = (x - 1)$ so $G(1) = 0$. Since we want W to be a multiple of G, we will also have $W(1) = 0$. But we need to further encode into W the fact that, at the value of the gate (i.e. $x = 1$), the two inputs are being multiplied to create the output. As we saw above, at the first gate, we have:

$$C_1 * C_1 - C_2 = 0$$

This encodes what happens at the gate, and it equals zero. Since we need $W(1) = 0$, maybe $W(1)$ looks like this:

$$W(1) = C_1 * C_1 - C_2 = 0$$

So this suggests to us a way for our C values to be part of the definition of W, structuring them so they reflect the multiplication gates they're connected to. Interesting.

Let's go further and introduce a second gate. We assign the second gate the value

2. Remember we're trying to work up to our original problem of 4 gates that defines $C_5 = C_1 * C_1 * C_1 * C_1 * C_1$. With one gate we're computing $C_2 = C_1 * C_1$. With a second gate we can compute $C_3 = C_2 * C_1 = C_1 * C_1 * C_1$. Since we assigned our gates the values of 1 and 2, the $G(x)$ for this 2 gate system should look like $G(x) = (x-1)(x-2)$, and since $W(x)$ needs to be a multiple of $G(x)$, it must also equal 0 at $x = 1$ and $x = 2$. Following our logic from a single gate, maybe we can structure $W(x)$ so that $W(1) = C_1 * C_1 - C_2 = 0$, reflecting the structure of the first gate, and so that $W(2) = C_2 * C_1 - C_3 = 0$, reflecting the structure of the second gate. But how do we do this?

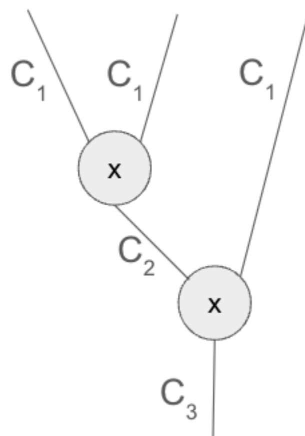

Figure 2: Simplified circuit with only two multiplication gates

Let's look at those two side by side to get some ideas:

- $W(1) = C_1 * C_1 - C_2 = 0$
- $W(2) = C_2 * C_1 - C_3 = 0$

Hmm. It seems the structure of $W(x)$ should somehow look like $W(x) = L * R - O$ whenever x is the value of a gate. But since $W(x)$ is a polynomial, we need L, R, O to also be polynomials. So maybe $W(x)$ can actually be structured in terms of three sub polynomials like this:

Encoding the Wires

$$W(x) = L(x) * R(x) - O(x)$$

Then, when x is equal to the value of a gate, $L(x)$ is equal to the left wire of the gate, $R(x)$ is equal to the right wire, and $O(x)$ is equal to the output wire.

Feels like some progress here! Sometimes when you want to make a big thing that encodes a bunch of pieces, you start by making a smaller thing for each piece and then combine them. This is exactly what's happening here. Instead of just trying to figure

out some big $W(x)$, we're realizing there's probably some sub structure, where $W(x)$ is made up of smaller polynomials $(L(x), R(x), O(x))$ that reflect the structure of what we're trying to do. We still need to figure out the structure of these smaller polynomials. Maybe we have to break them down even further.

4.3 Encoding Individual Wires

It seems that each of our new polynomials $L(x)$, $R(x)$, and $O(x)$ needs to encode all the wires, since any wire might be a left input, right input, or output to any gate. Let's take $L(x)$ and think about what that could look like. We know that for each value of x that corresponds to a gate, $L(x)$ needs to give us the correct C_i that is the left value for that gate. So $L(x)$ must contain all the C values. And only Philip knows all the C values.

In our example, at gate 1 the left input is C_1, and at gate 2 it's C_2. So $L(1) = C_1$ and $L(2) = C_2$. Can we define $L(x)$ so it returns the correct C_i when x is a gate? It's almost like for each C_i we need a sub-function to multiply it by that's either 1 or 0 at the value of each gate depending on whether that C_i is a left input or not.

Let's try to make this concrete. We need $L(x)$ to combine all the different C_i values. When x is the value of a gate (in our simple example, 1 or 2), $L(x)$ must return the C_i that is the left input for that gate. Lets introduce a sub-function $L_i(x)$ for each C_i defined so that for each value of x that is a gate, $L_i(x)$ is 1 if C_i is a left input of that gate, and it is 0 otherwise. Then we could just define $L(x)$ by multiplying each C_i by its $L_i(x)$ and adding them up:

$$L(x) = C_1L_1(x) + C_2L_2(x) + C_3L_3(x)$$

We call this kind of combination, where values (our $L_i(x)$) are multiplied by coefficients (our C_i) and then added up, a *linear combination*. With this linear combination and the way we've defined the L_i, when x is a gate, all the L_i are 0 except the one where C_i is a left input to that gate. So L will return just the value of that wire, C_i. So for $L_1(x)$, which encodes when C_1 is a left input, $L_1(1) = 1$ because C_1 is a left input to gate 1, and $L_1(2) = 0$ because C_1 is not a left input to gate 2. For $L_2(x)$, which encodes where C_2 is a left input, $L_2(1) = 0$ since C_2 is not a left input to gate 1, and $L_2(2) = 1$ since C_2 is a left input to gate 2. And finally, $L_3(x) = 0$ since C_3 is not a left input to any gate. Thus $L(1) = C_1$ and $L(2) = C_2$, as desired.

Now we really might be getting somewhere. We can do the exact same thing for $R(x)$ and for $O(x)$. And since we're supposed to have $W(x) = L(x)R(x) - O(x)$, we can put it all together for our 2 gate system like this:

$$W(x) = \Big(C_1L_1(x) + C_2L_2(x) + C_3L_3(x)\Big) * \Big(C_1R_1(x) + C_2R_2(x) + C_3R_3(x)\Big)$$

$$-[C_1 O_1(x) + C_2 O_2(x) + C_3 O_3(x)]$$

This polynomial $W(x)$ encodes the entire computation of our simple 2 gate system, $y = x^3$ (i.e. $C_3 = C_1 * C_1 * C_1 = C_2 * C_1$). Each wire i gets three sub-polynomials $L_i(x), R_i(x), O_i(x)$, which are either 0 or 1 when x is the value of a gate (encoding the wires relationship to that gate). This actually ensures that if the intermediate C_i values were computed correctly, then $W(x)$ is 0 at every gate, since $L * R - O = 0$. And since $G(x)$ was defined to be 0 at every gate, $W(x)$ will be a multiple of $G(x)$. Splendid.

So encoding each computation as a polynomial really comes down to defining these 3 sets of sub-polynomials, the $L_i(x), R_i(x), O_i(x)$. Notice that these sub-polynomials only depend on the structure of the gates, and not on the actual values of the wires. So even though with $W(x)$ we are encoding the full computation with specific values for the wires, i.e. $C_3 = C_2 * C_1 = C_1 * C_1 * C_1$, the set of sub-polynomials encodes the computation itself (i.e. two multiplications), without the specific values of the wires.

What do these sub-polynomials look like? For each C_i we have three of them, $L_i(x), R_i(x), O_i(x)$. We define them as being 0 or 1 when x is the value of a gate, and we don't really care what they are for any other value of x. So each one is defined by a set of points (x, y) where x is the value of a gate (1, 2, etc.) and y is either 0 or 1, depending on whether C_i is the corresponding input/output at that gate. Notice that this is the second method of encoding values in a polynomial we covered in Section 3.1. A set of n points uniquely defines a polynomial of degree $n - 1$. We have one point for every gate.

Let's lay it all out for our simple case. We have two gates, so each sub-polynomial is defined by two points, one for each gate, and is thus a line. For instance, let's take C_1 and the sub-polynomial that encodes where it's a left input, $L_1(x)$. We know C_1 is a left input to gate 1 but not to gate 2. So $L_1(1) = 1$ and $L_1(2) = 0$. In other words, $L_1(x)$ is defined by the two points $(1, 1)$ and $(2, 0)$. We can work out from $y = mx + b$ that this corresponds to $L_1(x) = -x + 2$. We can do the same for the other sub-polynomials.

Let's look at what this looks like for all wires and sub-polynomials in our simple 2 gate system. We have a point for each gate telling us if the wire is a left input, right input, or output for that gate, and from the two points we can derive an equation for the sub-polynomial. Starting with C_1, which is a left input to gate 1 and a right input to gate 1 and 2:

C_1	**Gate 1**	**Gate 2**	**Equation**
$L_1(x)$	$(1,1)$	$(2,0)$	$L_1(x) = -x + 2$
$R_1(x)$	$(1,1)$	$(2,1)$	$R_1(x) = 1$
$O_1(x)$	$(1,0)$	$(2,0)$	$O_1(x) = 0$

Notice that $R_1(x)$ is just the constant 1 because C_1 is a right input to every gate, and $O_1(x)$ is just 0 always since C_1 is never an output of any gate. How about C_2, which is an output of gate 1, and a left input to gate 2:

C_2	**Gate 1**	**Gate 2**	**Equation**
$L_2(x)$	$(1,0)$	$(2,1)$	$L_2(x) = x - 1$
$R_2(x)$	$(1,0)$	$(2,0)$	$R_2(x) = 0$
$O_2(x)$	$(1,1)$	$(2,0)$	$O_2(x) = -x + 2$

This time we have $R_2(x) = 0$ because C_2 is not a right input to any gate. Notice also the $O_2(x)$ is the same as $L_1(x)$ because C_2 is an output to the same gates that C_1 is a left input (just gate 1). We also have C_3, which is just an output of gate 2:

C_3	**Gate 1**	**Gate 2**	**Equation**
$L_3(x)$	$(1,0)$	$(2,0)$	$L_3(x) = 0$
$R_3(x)$	$(1,0)$	$(2,0)$	$R_3(x) = 0$
$O_3(x)$	$(1,0)$	$(2,1)$	$O_3(x) = x - 1$

Again we have $L_3(x) = 0$ and $R_3(x) = 0$, because C_3 is not a left input or a right input for any gate. And here we see that $O_3(x)$ is the same as $L_2(x)$, because C_3 is output to the same gates that C_2 is a left input (just gate 2).

So that's that. In general with n points (gates) we'll see something like this. The highest degree among the sub-polynomials will be $n - 1$. Some of them will be the same. Some will be 0. Combined together into $W(x)$, they define the structure of a computation.

Since we only have 2 gates, our sub-polynomials are all just linear functions (lines), and we can solve $y = mx + b$ to figure them out. But if we had more gates, we'd have higher degree functions here, and we could do what's called *polynomial interpolation* to figure out what polynomial these points correspond to. For instance, in our original problem of $y = x^5$, which has 4 gates, C_1 is a left input to only gate 1, and not to the other 3 gates. So $L_1(x)$ is defined by the points $(1,1), (2,0), (3,0), (4,0)$. With polynomial interpolation, this actually corresponds to the polynomial $L_1(x) = -\frac{x^3}{6} + \frac{3x^2}{2} - \frac{13x}{3} + 4$.

I know this seems like a lot and maybe a bit silly, but we arrived here just by trying to make $W(x)$ a multiple of $G(x)$ that reflects the structure of the circuit and it seems like we've actually pulled it off! And in the process we discovered this fun way to encode the structure of the gates into these sub-polynomials. We simply define each sub-polynomial in terms of a set of points that encode whether the corresponding wire is a left input, right input, or output at each gate!

Encoding All Wires in a Polynomial

We can now write a general version of this for any number of multiplication gates. The symbol $\sum_i$ means we "sum" over all the values of i (i.e. for each wire):

$$W(x) = (\sum_i C_i L_i(x))(\sum_i C_i R_i(x)) - \sum_i C_i O_i(x)$$

If you were to see this equation out of nowhere, it would probably be kind of menacing, like what does this have to do with anything. But now that we've derived it for ourselves by thinking through how we can simply encode the circuit into a polynomial $W(x)$ that's a multiple of the polynomial $G(x)$, it makes a lot more sense!

5 Returning to a Proof

We've come a long way from where we started, but we still have a ways to go. Remember we're still trying to enable Philip to prove to Veronica that he did the computation correctly, without her having to repeat it.

Our idea was to reduce this to a division problem. We'd encode the circuit into polynomials $W(x)$ and $G(x)$ and have Philip divide them to get $H(x)$. Veronica could then check the division by doing a multiplication at a single point s, $W(s) = G(s)H(s)$. We figured out in the last section that we could encode the computation in $W(x)$ so it is a multiple of $G(x)$ and can thus be divided. We saw that we can label the gates and encode them as $G(x)$. And we can further encode the structure of the computation (how wires are combined into gates) in these sub-polynomials $L_i(x), R_i(x), O_i(x)$ for each wire. And this can all be known by both Veronica and Philip. But then Philip can take the sub-polynomials, multiply by the C_i's (which only he knows) to get $W(x)$. Since $W(x)$ will be 0 at the same places $G(x)$ is 0, he can divide by $G(x)$ to get $H(x)$, evaluate at Veronica's secret s, and send her $W(s)$ and $H(s)$. She can check a single multiplication $W(s) = G(s)H(s)$ Amazing.

But if she gives Philip the secret point s so he can compute $W(s)$ and $H(s)$ for her, she has no way to know he gave her the real $W(s)$ and $H(s)$. He could have just made up some value h, computed $w = G(s) * h$, and sent Veronica back w and h, and she wouldn't be any wiser. So now we need a way for Veronica to ensure that the value she got back from Philip actually came from the polynomial $W(x)$ and he isn't just trying to trick her. Can we even do this?

In the previous section we figured out how to construct $W(x)$ in terms of $L(x)$, $R(x)$, $O(x)$, with all the C_i and the sub-polynomials. Now we need Philip to prove he really constructed $W(x)$ like this. Lets try to break down all the things Philip needs to prove into a few distinct components, since maybe that will help us figure out how he can prove it all and how Veronica can verify it all:

Summary: What Needs to be Proven

1. $W(x)$ is a multiple of $G(x)$, i.e. there is some $H(x)$ such that $W(x) = G(x)H(x)$
2. $W(x)$ is constructed as $W(x) = L(x)R(x) - O(x)$
3. Each of $L(x)$, $R(x)$, and $O(x)$ are constructed as
   (a) some linear combinations of their underlying sub-polynomials
   (b) the same linear combination using the same coefficients C_i

   i.e. $L(x) = \sum_i C_i L_i(x)$, $R(x) = \sum_i C_i R_i(x)$, $O(x) = \sum_i C_i O_i(x)$

Note that we separated Point 3 into two sub-points. Point 3a is saying that each of the L, R, O polynomials is made up of its sub-polynomials in some way, while Point 3b says they're made up by their sub-polynomials in the *same* way. While it might be possible to prove both of these things at once, it might also help us get a better sense of what we need to do to prove them separately.

Let's start with point 3a, since it deals with the smallest sub components of our polynomials. And lets start by just considering $L(x)$, since anything we do for it we can apply to $R(x)$ and $O(x)$ as well. We need some way for Philip to compute $L(s)$ where Veronica can easily verify it came from some underlying linear combination of $L_i(s)$, i.e. $L(s) = \sum_i c_i L_i(s)$. The problem we keep having when Philip gives Veronica values is he could just make them up, and we have no way to ensure they came from applying a specific polynomial to a value given by Veronica. Maybe there's a way we can encode or encrypt those values so that it's easy to check that they could only have been computed from a specific polynomial? Since polynomials are just repeated addition, multiplication, and exponentiation, what we need is a way for Philip to prove he's doing these kinds of operations.

5.1 Math on Encrypted Data

One idea could be to try and find a way for Philip to do some math on numbers he can't actually access himself where the only way he could get the result is to do a specific operation so that he can prove he did things correctly and isn't making stuff up. Is that even possible? It's basically like we need some way to operate on encrypted numbers. Doing this kind of thing in general is very hard and known as "fully homomorphic encryption". Maybe you could have invented fully homomorphic encryption too but lets finish inventing SNARKs first. Fortunately there are more limited ways to compute on encrypted data that are well known and easier to understand. What we need is a way to

transform numbers into new numbers, such that we can still do operations on the original numbers, but where undoing the transformation is very hard.

We already talked about things that are easy to do one way and hard to reverse. It's easier to multiply than to divide. And it's easier to raise something to an exponent than to take the logarithm. But maybe you also remember from high school that even though logarithms are hard, there are actually "exponent rules" that allow you to do some math on things in the exponent. Wait, what? Does this mean we can basically do encrypted computations with nothing but high school math? Pretty much.

Lets remind ourselves of the exponent rules. They're pretty simple. Say we have some base g and we raise it to the x so we have g^x. And say we have another value y we can get g^y. If we multiply these two, we add the exponents: $g^x * g^y = g^{x+y}$. And if we have g^x and we raise this to the power of y, we multiply the exponents: $(g^x)^y = g^{x*y}$. To summarize:

Summary: Exponent Rules

- 1. $g^x * g^y = g^{x+y}$
- 2. $(g^x)^y = g^{x*y}$

Those are all the exponent rules. If we assume taking logarithms is hard, then raising g to the power of x is a way of encrypting x, and if we do this with x and y, we can actually do addition on the encrypted values of x and y. Cool!

Notice though there's a limitation. If we have two encrypted values, g^x and g^y, there's no easy way to actually multiply x and y. So while we're able to do "homomorphic addition" like this, we can't actually do "homomorphic multiplication". This is pretty annoying, but hopefully we can do without it. We'll see.

Now, it's important to point out here that this kind of encryption we're talking about (encrypting x by doing g^x) isn't secure in general. There's lots of algorithms to do logarithms quickly, even if they're hard to do in your head, and so this would definitely not be secure. But fortunately, there's an easy way to make logarithms really hard, even for computers: use large finite fields. We mentioned finite fields in passing before, but they're just the set of integers with a finite size, where you use "clock math". Say we take the field of size 12, that's the integers 0 to 11, and if you add or multiply above 11 you just go back around. For instance, on a clock, 5 hours after 9 o'clock is 2 o'clock, so $9 + 5 = 2$. We can also multiply, for instance, $5 * 4 = 8$. This kind of "clock math" is also known as "modular arithmetic" from the "modulo" operation, which has the symbol %. You might remember seeing like $5 * 4\%12 = 8$, which says that the remainder after dividing $5 * 4$ by 12 is 8.

The point is, if you use a *really large* finite field (really large modulus), then logarithms

become *really* hard, even for computers - the so-called *Discrete Logarithm Problem*. This is what a lot of our cryptographic signature schemes depend on. You may have also heard of elliptic curves. These are just variations on modular arithmetic in finite fields where logarithms are *even harder*. So hard we pretty much trust they're impossible without quantum computers and build most of our internet and blockchain security based on that assumption. So we can use those elliptic curves here as well. We won't go into the details of how elliptic curves work. Maybe you could have invented those yourself too, but we'll leave that for another time. For now, it's enough to know that elliptic curves define a *finite* set of "numbers" where you can do the exponent rules we know and love, and where logarithms are basically impossible. So you can securely encrypt a number x by doing g^x on the elliptic curve. Pretty cool stuff.

So let's go back to our problem. We want Philip to first prove he's giving values that actually come from combining certain polynomials, rather than just making up random numbers. How can we do that? Let's start as simple as possible. Suppose we have two simple polynomial $a(x) = x$ and $b(x) = x^2$. We can make a linear combination of these two polynomials, let's call it $c(x)$, by introducing coefficients (c_1 and c_2) for each and adding: $c(x) = c_1a(x) + c_2b(x) = c_1x + c_2x^2$. Now if we want Philip to compute $c(s) = c_1s + c_2s^2$, how can he prove to us it came from combining $a(s)$ and $b(s)$?

Well, let's see if our exponent encryption scheme can help us. What if we encrypted the values s and s^2. We would get new values g^s and g^{s^2}. To Philip, these are just numbers (since he doesn't know what s is). He could raise each of these numbers to their respective coefficient and multiply the result, to get:

$$(g^s)^{c_1}(g^{s^2})^{c_2} = g^{c_1s+c_2s^2}$$

But now what? Philip computed $c(s)$ on encrypted data ("in the exponent"), but how do we prove that this result actually came from taking the original encrypted values, g^s and g^{s^2}, raising them each to a coefficient (c_1 and c_2), and multiplying them?

5.2 Proving Knowledge of a Secret Exponent

We can try to simplify the question a bit further. Suppose Philip has a secret value p. He can encrypt it by raising g to p: g^p. But how can he prove he actually knows a value p and that he raised g to the p without revealing p? If Philip just gives the two values g and g^p to Veronica, she has no way to know that the second value is actually the first value raised to any power, let alone p. She just sees two numbers. This is basically the same problem we have above, since p here is basically the coefficient we raise the polynomials to before multiplying them (c_1 and c_2 above). So if we can figure this out, Philip can prove he's linearly combining polynomials, as we require.

The problem is that with just the numbers g and g^p alone, there's not much Veronica

can do. But maybe if she comes up with some additional values first, there will be more structure we can work with. We can try to use symmetry, which is commonly helpful in figuring things out. Just like Philip has his secret p value, lets give Veronica her own secret value v, and have her raise g to that value: g^v. What can we do from here?

Well we know from the exponent rules that we can combine a number with an encrypted number. Suppose Veronica sends her encrypted value, g^v, to Philip. Without knowing v, Philip can raise it to p. This would give $(g^v)^p = g^{vp}$. Veronica could do the same thing by getting the number g^p from Philip and raising it to the v, giving the same result: $(g^p)^v = g^{pv} = g^{vp}$. So now even though only Veronica knows v and only Philip knows p, they were both able to compute g^{vp}.

Does this prove that Philip knows p without revealing p? Veronica sent Philip two values, $(g,\ g^v)$. Philip took these values and raised them both to the p, giving two new values $(g^p,\ g^{vp})$. He doesn't know v, only Veronica does. Veronica can take the first value she got from him and raise it to v to see if it equals the second value. If it does, it means Philip produced two values where one is the other raised to a power that he doesn't know (i.e. v). In other words he produced two values (h, h^v), where he doesn't know v. But given logarithms are so hard, and we only have our exponent rules to work with, it seems like the only way he could possibly do this is if he started with two other values (g, g^v), and raised them both to the same exponent p, giving (g^p, g^{vp}). If we let $h = g^p$, then this corresponds to (h, h^v), as above. Thus, by having Veronica check these two values, Philip has successfully proven he knows p and that he took the two value from Veronica and raised them both to the power of p. Wow!

This phenomenon is actually known in the literature as the *knowledge of exponent assumption*. It's a key part to making these SNARKs work, though it's actually a bit controversial itself, because it's technically not falsifiable (there's no formal way to prove someone knows something). Hence, it is an assumption. For our purposes, assuming logarithms are hard, it seems to do the job for us, and we were able to come up with it just by trying to figure out how Philip can prove to Veronica he knows some value without revealing it, and using our exponent rules. Go us!

Let's summarize the new machinery we discovered for making our proof about $W(x)$:

Summary: Encrypted Math

- we can "encrypt" value x as g^x for some base g
- we can add encrypted values, and we can multiply an encrypted by an unencrypted value, but we can't multiply two encrypted values
- this is all secure if the operations are done on special "numbers" defined by elliptic curves

Summary: Knowledge of Exponent

Philip can prove he knows a secret p without revealing it. Veronica takes her secret v and sends (g, g^v) to Philip. He raises both to the p and sends them back, (g^p, g^{vp}). Veronica takes the first of Philip's two value and raises it to v, which only she knows, and if it equals the second value then Philip must know some value p that he raised both her original values to. Hence he has knowledge of the secret exponent p.

Remember the whole reason we came up with the knowledge of exponent test was because we were trying to help Philip prove he knows a polynomial constructed as a linear combination of sub-polynomials. Let's put it to work.

6 Breaking up the Proof

Armed with these new powers (encrypted math and the knowledge-of-exponent test) let's get back to our original business of trying to prove the main points we laid out at the start of Section 5 of what needs to be proven about how $W(x)$ is constructed. We had started with point 3a by trying to prove to Veronica that Philip computed $L(s)$ as a linear combination of the sub-polynomials $L_i(s)$, i.e. $L(s) = \sum_i c_i L_i(s)$. This led us to developing our basic encryption scheme and the knowledge-of-exponent test. Let's now try to put them to use.

So Philip has a bunch of these sub polynomials $L_i(x)$. He needs to take each one, multiply it by a coefficient, and then add them all together. Ultimately, those coefficients will be the different values C_i, some of which Veronica knows (the inputs/outputs), but most of which she shouldn't know (the intermediate values).

6.1 Proving a Linear Combination

How can we prove that $L(s)$ was computed by linearly combining a set of $L_i(s)$ using coefficients that correspond to the values of the wires? We don't really want Veronica to have to have to handle all the $L_i(x)$ polynomials for each i, but in principle she could, since they just encode the underlying computation, and they don't include the intermediate values for a specific computation (i.e. they don't include any of the C_i which we don't want her to know). Actually, she would only have to compute all the $L_i(s)$ for each i *once* for the given computation, and could reuse them many times to check the same computation for different values of C_1. So if Veronica does this work one time, say for the function $y = x^5$, then Philip will be able to keep making proofs to her about $y = x^5$ for many different values of x (i.e. many different values of C_1). So even though this setup step might be expensive, it can be infinitely reused. That's still useful. Note though that

if someone else, say Valerie, wanted to verify the same computation, she'd also have to do her own setup using her own secret value s_2. We'll come back to the problem of doing a shared (or "trusted") setup, that any one can use, to make the protocol *non-interactive.*

So Veronica knows all the $L_i(x)$ and is ready to do the setup. She selects the single secret value s for evaluating the polynomials at (thank you Schwartz–Zippel) and computes $L_i(s)$ for each i. She also needs a secret value for the knowledge of exponent test. In the previous section we called Veronica's knowledge of exponent secret v and Philip's p. Philip was trying to prove he knows p. But in reality Philip is going to be proving he knows the C_i, which Veronica doesn't know. So let's give Veronica's secret a new name too. We'll call it α because that's cool. Veronica then computes all the encrypted values $g^{L_i(s)}$ and $g^{\alpha L_i(s)}$ for each i. Having that second value for each i with the α in the exponent allows her to set up our knowledge-of-exponent test.

Now she sends these to Philip. Note these values are just numbers, so Philip has no idea what the underlying values are in the exponent, since it's hard to do the logarithm. Philip then takes each term and raises it to the power of the relevant coefficient C_i, so we get $g^{C_i L_i(s)}$ and $g^{\alpha C_i L_i(s)}$ for each i. And now for each of these two sets of terms, he can multiply all the terms together to add the values in the exponent, e.g. if he multiplies $g^{C_1 L_1(s)}$ with $g^{C_2 L_2(s)}$ he gets $g^{C_1 L_1(s)+C_2 L_2(s)}$. Doing this for all the i actually just computes the value for the full polynomial $L(s)$ since $L(x) = \sum_i C_i L_i(x)$. So after multiplying for each i Philip ends up with $g^{L(s)}$ and $g^{\alpha L(s)}$. Cool!

He can send these two numbers back to Veronica and she can check that indeed one of them is the other one raised to the power of α, and since Philip doesn't know α, this *must* mean (according to knowledge-of-exponent) that Philip knows some coefficients and used those to combine the terms Veronica gave him before, exactly as we hoped! This is amazing. Philip has now proven that he knows a bunch of coefficients (the C_i's) and he can linearly combine a bunch of encrypted values according to those coefficients. Awesome!

What about R and O? We said whatever we do for L should work for R and O too, which is true, but with a catch. If we (as Veronica) use the same secret exponent value α for all of them, then we might be in trouble. For instance, if we only send Philip all the $g^{L_i(s)}$ and $g^{\alpha L_i(s)}$ and he sends back $g^{L(s)}$ and $g^{\alpha L(s)}$ we know that the $L(s)$ in the exponent must be a linear combination of the $L_i(s)$'s that we started with, since that's the only thing Philip has to work with that was also raised to the power of α. But if we also send him all the $g^{R_i(s)}$ and $g^{\alpha R_i(s)}$, then he's free to mix and match different L_i with R_i and could send us back something like $g^{K(s)}$ and $g^{\alpha K(s)}$ where, for instance $K(s) = L_1(s) + R_1(s)$. This would still pass our knowledge of exponent check but it's basically nonsense, because it fails to ensure he only combines L_i polynomials with other L_i polynomials (for different values of i) and R_i polynomials with other R_i polynomials.

This seems simple enough to solve though. Things worked when we were just dealing

with the L_i's, and the only reason adding R_i's is a problem is because we used the same secret value α for both. So instead of using the same secret value α, we use a different value for each family of polynomials: α_L for the L_i, α_R for the R_i, and α_O for the O_i. That way, Philip won't be able to mix and match because they all use different secret exponents that he doesn't know so the check will fail. The only way the check will pass is if he combines L_i's with L_i's and R_i's with R_i's, as we'd like.

Excellent. So with the above, we've pretty much solved point 3a of the points we need to prove. We mentioned before that point 3b is very similar, but where point 3a just proves that each of $L(x)$, $R(x)$, and $O(x)$ are linear combinations of sub-polynomials, point 3b is about proving that they're all the same linear combination, using the same coefficients. So how can we prove that?

6.2 Proving *the same* linear combination

Note that Philip has already given us the values $g^{L(s)}$, $g^{R(s)}$, $g^{O(s)}$. He already proved in the last subsection that each was created through a linear combination, but now we need to prove they were each created by the *same* combination (i.e. using the same set of C_i). So we probably need some way to combine these encrypted values. One thing we can do is multiply all the encrypted values together: $g^{L(s)}g^{R(s)}g^{O(s)}$. By the exponent rules, the values in the exponent all get added up: $g^{L(s)+R(s)+O(s)}$ Does that give us anything?

Looking at just what's in the exponent, we expect Philip computed:

$$L(s) = C_1L_1(s) + C_2L_2(s) + ... + C_5L_5(s)$$

$$R(s) = C_1R_1(s) + C_2R_2(s) + ... + C_5R_5(s)$$

$$O(s) = C_1O_1(s) + C_2O_2(s) + ... + C_5O_5(s)$$

Notice they all have the same structure, where the first term is multiplied by C_1, the second by C_2, and so on. So if we added them all together like in $g^{L(s)+R(s)+O(s)}$, we'd be able to factor out those coefficients:

$$L(s) + R(s) + O(s)$$

$$= C_1L_1(s) + ... + C_5L_5(s) + C_1R_1(s) + ... + C_5R_5(s) + C_1O_1(s) + ... + C_5O_5(s)$$

$$= C_1(L_1(s) + R_1(s) + O_1(s)) + ... + C_5(L_5(s) + R_5(s) + O_5(s))$$

Interesting. So we multiplied these three encrypted values and that added them in the exponent:

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

And in principle we should be able to factor the coefficients out of that sum. But since all of this stuff is encrypted in the exponent, we can't actually get access to it to see if we can factor out the coefficients. That said, we can get Philip to do another knowledge-of-exponent proof for us with this combined term! Veronica will have to pick another secret value for a new knowledge of exponent check, let's call it β this time. Since Veronica has to compute all the $g^{L_i(s)}$, $g^{R_i(s)}$, $g^{O_i(s)}$ as part of the setup, she can also compute one more term for each i by multiplying them all together and raising them to β. So for each i she'll also compute the term:

$$(g^{L_i(s)}g^{R_i(s)}g^{O_i(s)})^\beta = g^{\beta L_i(s)}g^{\beta R_i(s)}g^{\beta O_i(s)} = g^{\beta(L_i(s)+R_i(s)+O_i(s))}$$

If she also sends these terms to Philip, he can raise each one to the appropriate C_i and multiply everything together, computing $g^{\beta(L(s)+R(s)+O(s))}$ without knowing β. Since Veronica knows β, and since Philip also sent her each of $g^{L(s)}$, $g^{R(s)}$, $g^{O(s)}$, she can take those, multiply them, and raise them to the β, and check that the result equals the $g^{\beta(L(s)+R(s)+O(s))}$ term she got from Philip. If so, it seems this would prove that Philip did in fact use the same C_i to combine each of the L_i, R_i, and O_i. Tada!

But wait a minute. When we did the knowledge of exponent test with the α above, we ended up needing to use three different secret values, $\alpha_L, \alpha_R, \alpha_O$, to force Philip to only combine L_i terms with other L_i, and so on. Are we going to have a similar issue with the β? It's not quite the same because in the previous case with only one α, we had different terms raised to it like $g^{\alpha L_i(s)}$ and $g^{\alpha R_i(s)}$, which would have allowed Philip to combine L_i's with R_i's which would be no good. But here all the terms are combined, and Veronica is just providing $g^{\beta(L_i(s)+R_i(s)+O_i(s))}$. But there still might be some way for Philip to cheat (i.e. use different coefficients to combine the L/R/O sub-polynomials), so it might be a good idea to use multiple values, like $\beta_L, \beta_R, \beta_O$. If you think about it a bit, with only the one β, you can figure out a way for Philip to cheat when some values like $L_i(s)$ and $R_i(s)$ might happen to be the same. This works because when some of the encrypted values are the same it can hide the fact that he used different coefficients, and so trick Veronica.

So that means Veronica will have to create three more secrets $\beta_L, \beta_R, \beta_O$ and instead of sending Philip the terms above using only the one β, she'll have to send him:

$$g^{\beta_L L_i(s)}g^{\beta_R R_i(s)}g^{\beta_O O_i(s)}$$

He wouldn't be able to notice the difference (since he can't see what's in the exponent), and he'll still just combine them for all the i by raising them to the C_i, multiplying, and sending the result to Veronica. But this will allow Veronica to prevent him from any possibility of cheating by using different coefficients to combine the sub-polynomials. She'll take each of the $g^{L(s)}, g^{R(s)}, g^{O(s)}$ she got from him before, raise them to the appropriate

β value, and multiply them all, and check if it equals the β term he sent back. If so, it proves to her he used the same coefficients to combine all the sub-polynomials, with no way to cheat. Yay!

We're making really great progress here. We've now got proofs that the L, R, and O polynomials are made up of linear combinations of the sub polynomials using some set of coefficients, and we've got a proof that they all use the *same* linear combination, i.e. the same set of coefficients. That set of coefficients is of course our C_i - the actual values of the wires in the circuit. But we still need to check that they're actually the right values, and the way we do that is via the divisibility check. Remember our old friends $W(x)$ and $G(x)$? We haven't seen them in a while because we've been working with the sub-components of $W(x)$ (the $L(x)$, $R(x)$ and $O(x)$). But it's time to bring them back.

If you recall our key points to prove from Section 5, we've proven points 3a and 3b (about the L, R, O polynomials), but we still have points 1 and 2 left (about $W(x)$ itself). Point 1 was the divisibility check, the ultimate check that $W(x)$ is a multiple of $G(x)$, in other words, $W(x) = G(x)H(x)$. And Point 2 was about ensuring that $W(x)$ itself is built from the $L(x)$, $R(x)$, and $O(x)$ polynomials as $W(x) = L(x)R(x) - O(x)$. Since these are both equations for $W(x)$, maybe we can actually combine them by setting them equal to each other. Doing so gives us:

$$L(x)R(x) - O(x) = G(x)H(x)$$

But what do we do with this? How can Veronica even check this?

7 Completing the Proof

So far, we've been doing all our math "in the exponent" so its encrypted. We've done a bunch of work with the L_i, R_i, and O_i in the exponent, in order to ensure that Philip wasn't cheating, so Veronica could verify he must have combined those small polynomials properly into the larger ones. In doing so, Philip was forced to compute the values $g^{L(s)}$, $g^{R(s)}$, and $g^{O(s)}$, without knowing s or what's in the exponent. We can't let Philip know s, since that could compromise all the proving we've done so far, so we have to continue doing our math "in the exponent".

So we already have L, R, and O in the exponent. Veronica can compute $G(x)$ for herself (it's just the gates, which she knows) and she knows s, so she can also compute $g^{G(s)}$. But she doesn't know $H(x)$, only Philip does, and Philip doesn't know s, only Veronica does. So how can Philip compute $g^{H(s)}$?

7.1 Evaluating an Encrypted Polynomial

So far, anything that's required computing directly with s in the exponent has been done by Veronica. But generally we've been trying to avoid her having to know the results of specific computations. While $G(x)$ reflects a set of gates, and while the $L_i(x)$, $R_i(x)$, and $O_i(x)$ reflects the general structure of the computation (in our case, x^5, for any value of x), $W(x)$ represents the specific computation carried out when $x = C_1$, including all the intermediate values that result. And similarly, $H(x)$ is the result of dividing $W(x)$ by $G(x)$, so it's also specific to that particular computation, including all the intermediate values. And intermediate values are the very thing Veronica wants to avoid.

So how does Philip compute $g^{H(s)}$? Well, $H(x)$ is a polynomial, and just like any polynomial, it can be represented as $H(x) = h_0 + h_1x + h_2x^2 + h_3x^3 + ...h_nx^n$. Recall this was the first way we saw of encoding numbers in a polynomial in Section 3.1. How does that help us here in evaluating $g^{H(s)}$? Both multiplication by a constant and addition are things we know how to do "in the exponent". And beside taking the main variable to higher and higher powers, a polynomial is just a bunch of multiplication by a constant and addition. So if we had the values of the powers of s in the exponent to start with, then Philip could use the h_i values he has and combine everything to compute $g^{H(s)}$. In other words, if Veronica pre computes g, g^s, g^{s^2}, g^{s^3}, ..., g^{s^n}, then Philip can take them and go:

$$g^{h_0}(g^s)^{h_1}(g^{s^2})^{h_2}...(g^{s^n})^{h_n} = g^{h_0+h_1s+h_2s^2+...+h_ns^n} = g^{H(s)}$$

and he would have it! Nice! We have everything we need in the exponent form. Now, getting back to $W(x)$. We know that $W(x) = L(x)R(x) - O(x) = G(x)H(x)$. We set $x = s$ so that Veronica only has to evaluate at a single point. Philip has computed $g^{L(s)}$, $g^{R(s)}$, $g^{O(s)}$, and $g^{H(s)}$ without knowing s. Veronica can compute $G(s)$ and $g^{G(s)}$ herself, but she doesn't actually know $L(s)$, $R(s)$, $O(s)$, and $H(s)$ directly, since she only has the encrypted versions from Philip. Now Veronica needs to take these encrypted terms from Philip and somehow check that $L(s)R(s) - O(s) = G(s)H(s)$. But doing so means Veronica needs to be able to multiply values in the exponent (both $L(s)R(s)$ and $G(s)H(s)$), while our exponent rules only let us add those values. This is the forbidden "homomorphic multiplication" that we mentioned earlier. Oh no!

7.2 Encrypted Multiplication

So what can we do here? This is the one part of the SNARK that you probably couldn't have come up with yourself. Or at least, you'd have to know this kind of thing exists and is possible. Remember how we said we're doing these exponents on special mathematical objects called elliptic curves? And we saw how you can do addition between two encrypted

values (exponent rule 1) and you can do multiplication between an encrypted value and an unencrypted value (exponent rule 2) but you can't actually do multiplication between two encrypted values? Well it turns out, there are special kinds of elliptic curves where you actually *can* do multiplication between encrypted values, but you can only do it *once*! What? What is this sorcery? These are a special kind of elliptic curve operation called "pairing" and they only work on specially defined "pairing" curves. Pairings are pretty complex, but actually allow you to do this sort of magic. The security of these elliptic curve "pairings" rests on stronger assumptions than just the Discrete Logarithm assumption we mentioned, but so far they seem to be secure. All we really need to know is that they exist and we can use them, and they allow us to do one single round of encrypted multiplication. Lucky for us, one round might be enough.

Let's call the pairing operation $ep()$, where "ep" stands for "elliptic-curve pairing". The way this operation works is it takes two encrypted values, say g^x and g^y, and it gives us the encrypted product of those two values, but with a different base than the one we started with. So where we started with base g, we're going to end up on say base j. We can think of this as a special new exponent rule. In other words:

Encrypted Multiplication: Elliptic Curve Pairings

With x encrypted as g^x and y encrypted as g^y, x and y can only be multiplied while encrypted using a an elliptic curve pairing operation:

$$ep(g^x, g^y) = j^{xy}$$

The result is in a new base, j, where normal exponent rules also apply.

So the encrypted values of x and y got multiplied into xy, and the result is still encrypted, but its encrypted to a different base (j) than the one we started with (g). The reason we can only do the magic multiplication operation once is that we end up on a different base, and there's no clear way to combine exponents from different bases. If we have j^a and g^b, there's no exponent rules we can apply. However if we have multiple things in base j, we can still use our exponent rules there, e.g. $j^a j^b = j^{a+b}$. But maybe that's ok, since one multiplication might be enough. How so?

Recall we had $W(x) = L(x)R(x) - O(x) = G(x)H(x)$. We know how to add things in the exponent, and we know how to do one-off multiplications now, but we don't know how to subtract things. Fortunately we can always rearrange a subtraction into addition: $L(x)R(x) = O(x)+G(x)H(x)$. Now it's in a form we can express using our exponent rules and our magic multiplication. Remember also that we already have all these polynomials evaluated at s "in the exponent" of g, i.e. we have:

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

We need to multiply $L(s)$ and $R(s)$. And we need to multiply $G(s)$ and $H(s)$ and add it to $O(s)$. Fortunately we can do all this using a few pairing operations:

$$ep(g^{L(s)}, g^{R(s)}) = ep(g, g^{O(s)})\,ep(g^{G(s)}, g^{H(s)})$$

Note we multiply $O(s)$ by 1 using the pairing so we can get it into base j so it can be added in the exponent. If we expand everything by applying our '*ep*' operation and our exponent rules it would look like:

$$j^{L(s)R(s)} = j^{O(s)} j^{G(s)H(s)} = j^{O(s)+G(s)H(s)}$$

which is just what we're looking for, though notice everything is in base j. Veronica will never actually be able to get the underlying encrypted values out, but she doesn't have to, she just needs to check for equality.

Wow! So we actually just managed to get Veronica to check that $W(x) = G(x)H(x)$ and that $W(x)$ was constructed a particular way as $W(x) = L(x)R(x) - O(x)$. And we did it all on encrypted data so that Philip couldn't cheat. And we managed to do it by checking only a single value at $x = s$ by relying on the Schwartz–Zippel lemma we discussed long ago. Pretty amazing stuff.

7.3 Separating input/output values from intermediate values

We have one final issue to solve. In all the excitement of trying to ensure Veronica doesn't have to know about any intermediate wires in the circuit's computation, we designed a system where she doesn't end up interacting with *any* of the wires, not even the input or output. But the whole point was that she provided an input (C_1) in order to get an output (C_5), so she must know them, otherwise Philip might be making a proof that he raised some number to the 5, but Veronica has no idea what!

So lets introduce some notation to distinguish between the wires (values of i) that Veronica knows from the ones she doesn't. She knows inputs and outputs. Let's call that IO. She doesn't know the wires for the intermediate values. Let's call those mid. Then we can refer to all the input/output C_i values as $\{C_i\}_{io}$. And we can refer to all the intermediate C_i values as $\{C_i\}_{mid}$. Using curly brackets like this is the standard way of referring to sets of numbers in math. So we can say Veronica only knows all the $\{C_i\}_{io}$, but Philip also knows all the $\{C_i\}_{mid}$. We can also break up our sub-polynomials into sets like this, for instance $\{L_i(x)\}_{io}$ and $\{L_i(x)\}_{mid}$. And we can name the combinations of these sets separately, so $L_{io}(x) = \sum_{io} C_i L_i(x)$, where we're only summing over values of i in IO. And similarly, $L_{mid}(x) = \sum_{mid} C_i L_i(x)$, where we're only summing over values of i in mid. Finally, of course, $L(x) = L_{io}(x) + L_{mid}(x)$

When Veronica is checking things from Philip, she needs to make sure that the correct

inputs and outputs are being used. So while Philip knows everything in IO, we don't actually need him to prove anything about the $\{C_i\}_{io}$ or $\{L_i(x)\}_{io}$, we can leave all that to Veronica. So when we talk about computing say $g^{L(s)}$, we'll have Veronica compute $g^{L_{io}(s)}$ using all the input/output values, and Philip will compute $g^{L_{mid}(s)}$ using all the intermediate values, and then Veronica will multiply the two to get:

$$g^{L_{io}(s)}g^{L_{mid}(s)} = g^{L_{io}(s)+L_{mid}(s)} = g^{L(s)}$$

And if we apply that everywhere, then actually, we're done!

8 Summarizing the Proof: a SARK

Let's try to summarize everything we've done, and then give a concise statement of the protocol in terms of what needs to be done to Setup, Prove, and Verify. We started by trying to prove a simple function like $y = x^5$. Veronica chose the value $x = C_1$ that she wanted to raise to the power of 5, and we called the result C_5. We know there are some intermediate values that Philip has to compute (C_2, C_3, C_4), but Veronica doesn't care about those. In order to get a handle on these intermediate values, we represented the computation as a graph, or circuit, which clearly shows us the values as 'wires' and the operations as 'gates'. We came up with the idea that if we could somehow represent this circuit as a division problem, that was hard to do but easy to verify, then maybe Philip could prove he had the correct intermediate values without Veronica having to know.

That led us to looking for the right kind of mathematical object to encode the circuit into. Since there are many values in the circuit (all the wires), we realized just using normal numbers wasn't sufficient, because we would lose a lot of the structure and information about the computation. Lucky for us, we remembered from high school that polynomials can be used to encode a large set of numbers, and that polynomials can also be added, subtracted, multiplied, and divided. We also saw the fortunate result that you could check if two polynomials were equal by just picking a random point and evaluating them both at that point. This seems like magic, but statistically, with big enough numbers, it works.

This led us to the idea to try and represent what Veronica knows (the gates) as a polynomial $G(x)$, and what Philip knows (the wires) as a polynomial $W(x)$, so that W would be a multiple of G if and only if it correctly encoded the values of the wires (i.e. correctly did all the computation). We then proceeded to encode the gates and wires into polynomials. For the gates, we simply numbered each gate in order $(1, 2, 3, 4)$, and encoded them as the roots of the polynomial

$$G(x) = (x - 1)(x - 2)(x - 3)(x - 4)$$

Since $G(x)$ is 0 at all these values, and $W(x)$ is a multiple of $G(x)$, W must also be 0 at all these values. And since at each multiplication gate we have that the left input times the right input minus the output equals 0, we were inspired to try to encode that into W using our C_i values. Thus we came up with a structure for W that looks like:

$$W(x) = L(x)R(x) - O(x)$$

where L, R, and O encode the left inputs, right inputs, and outputs for all the gates, and contain within them all the C_i's.

We then came up with a way to define L, R, and O so that when x is one of the gate values, they return the corresponding values of the wires (the C_i's). We realized we could do this with sub-polynomials like $L_i(x)$ for each wire C_i, where $L_i(x)$ is 1 when x is a gate that C_i is a left input for, and $L_i(x)$ is 0 for all other gates. And same for $R_i(x)$ and $O_i(x)$. This gave us a complete encoding of the wires into the polynomial $W(x)$.

With these polynomials in hand, we moved on to Philip trying to prove things, without being able to cheat. We needed him to prove three things (we started from simplest to most complex):

3. he constructed each of $L(x)$, $R(x)$, and $O(x)$ correctly by linearly combining all the sub-polynomials $L_i(x)$, $R_i(x)$, and $O_i(x)$ correctly for each i using the same values of the wires C_i

2. he constructed $W(x)$ as $W(x) = L(x)R(x) - O(x)$

1. he divided $W(x)$ by $G(x)$ yielding the quotient $H(x)$

To do this without Philip being able to cheat we needed all the math to be done on encrypted numbers, which we could do using our exponent rules from high school. We noted that it would really only be secure if we did the math in large 'finite fields' using 'elliptic curves', but we didn't have to worry too much about what that meant to understand how it all works. We could just proceed by thinking about exponents and exponent rules.

We then figured out a scheme for Philip to prove that he did a linear combination of encrypted values. This was called 'knowledge of exponent'. It worked by having Veronica create a secret value v, and sharing g and g^v with Philip. Philip then raised both of these to p, giving back g^p and g^{vp}. Veronica then checked that the second value Philip provided is the first raised to the v. Since Philip doesn't know v, this must mean he took the two values Veronica gave him and raised them both to the p, proving he knows a secret value p.

We used this knowledge of exponent for Philip to prove he combined the encrypted $L_i(x)$ polynomials into a single $L(x)$ by raising them to some coefficients (i.e. the C_i)

and multiplying. But we didn't want Veronica to have to operate on whole polynomials, so she picked a single secret point s, and performed an expensive setup computation that she would only have to do once where she computed $g^{L_i(s)}$ for all i and same for $R_i(s)$ and $O_i(s)$. And to do the knowledge of exponent checks, she also selected secret values $\alpha_L, \alpha_R, \alpha_O$ and computed $g^{\alpha_L L_i(s)}$ for each i and same for $R_i(s)$ and $O_i(s)$.

But she also needed to ensure that the L_i, R_i, and O_i were all combined using the same coefficients, namely the C_i, so she selected another set of secret values $\beta_L, \beta_R, \beta_O$ for another knowledge of exponent check and computed another set of terms as part of the setup that look like $g^{\beta_L L_i(s)} g^{\beta_R R_i(s)} g^{\beta_O O_i(s)}$.

Last but not least, to prove that $W(x) = G(x)H(x)$ for some $H(x)$ that Philip computes by dividing $W(x)/G(x)$, Philip needs to evaluate an encrypted $H(s)$ without knowing s. So as part of the setup, Veronica also computes all the terms g^s, g^{s^2}, ... g^{s^N} that Philip can then combine with the coefficients of $H(x)$ to actually compute $H(s)$ in the exponent without knowing s.

Vernoica sends all the setup terms to Philip. Philip computes $W(x)$ as he's supposed to and divides by $G(x)$ to get $H(x)$. He now constructs his proof using the encrypted values Veronica provided. Since Veronica knows the input and output wires, Philip only needs to operate on the intermediate wires. He takes all the knowledge of exponent terms Veronica gave him that correspond to intermediate wires (i.e. where i is in mid but not in IO. In our example, where i is $2, 3, 4$ but not $1, 5$). He raises them to the appropriate C_i and multiplies them together. He also takes the encrypted powers of s and raises them to the coefficients of H. And he sends this all back to Veronica.

Finally, Veronica can check everything. She completes the knowledge of exponent terms by computing them for the input and output wires (i.e. where i is in IO but not in mid. In our example, where i is $1, 5$ but not $2, 3, 4$). She multiplies these terms by Philip's terms to get the complete terms. She then checks the completed knowledge of exponent terms to see that Philip indeed did a linear combination of all the $L_i(s)$, $R_i(s)$, and $O_i(s)$, and that he used the same C_i for all of them. And then she checks that $G(x)H(x) = L(x)R(x) - O(x)$ by doing one final elliptic curve pairing check

$$ep(g^{L(s)}, g^{R(s)}) = ep(g, g^{O(s)})\, ep(g^{G(s)}, g^{H(s)})$$

And that's it, she's done!

Let's give a more concise statement of the protocol. There's three stages: Setup, Proving, Verification.

8.1 Setup

In the setup phase, Veronica pre-computes a bunch of things once for a given computation (a "circuit") that she can reuse to verify many proofs. In our case the computation we're interested in is $y = x^5$. Veronica can do the setup one time and then give Philip many different values of x (C_1) and for each one she can easily verify the result is correct. However the setup is expensive and requires her to first figure out the form of all the $L_i(x)$, $R_i(x)$, and $O_i(x)$, which can then be reused no matter what the value of C_1. Note that both Veronica and Philip will need to know the form of $G(x)$ as well as the $L_i(x)$, $R_i(x)$, and $O_i(x)$ polynomials.

So here's what Veronica has to do for the setup. First she does some work that Philip will have to do as well:

- Determine $G(x)$ based on the number of gates
- Determine $L_i(x)$, $R_i(x)$, and $O_i(x)$ for each i, based on the structure of the circuit for $y = x^5$

Then she does some work that only she will do, using secret values:

- Select a secret random value s to evaluate the polynomials at
- Compute $g^{L_i(s)}$, $g^{R_i(s)}$, $g^{O_i(s)}$ for each input/output i
- Compute $g^{G(s)}$
- Setup the knowledge of exponent checks:
  - Select secret random exponent values α_L, α_R, α_O, and β_L, β_R, β_O to test with
  - Compute $g^{L_i(s)}$ and $g^{\alpha_L L_i(s)}$ for each intermediate i
  - Compute $g^{R_i(s)}$ and $g^{\alpha_R R_i(s)}$ for each intermediate i
  - Compute $g^{O_i(s)}$ and $g^{\alpha_O O_i(s)}$ for each intermediate i
  - Compute $g^{\beta_L L_i(s)} g^{\beta_R R_i(s)} g^{\beta_O O_i(s)}$ for intermediate i
- Compute g^s, g^{s^2}, g^{s^3}, ... g^{s^N} where N is the degree of the polynomial $H(x)$ that Philip will compute (Veronica doesn't know $H(x)$, but she knows what degree it will be)

Note she only has to do the knowledge of exponent tests for the sub-polynomials associated with intermediate values, not for the input/output values, because she knows those. But she still has to compute the value of the sub-polynomials at s in the exponent for the input/output values.

8.2 Prove

With the setup complete, Veronica can pick a C_1 and ask Philip to compute the result and a proof. Philip can now take all the precomputed values from Veronica along with C_1 and get to work:

- Perform the computation (e.g. $C_5 = C_1^5$), computing all the intermediate values of C_i
- Determine $W(x)$ based on all the C_i, and $G(x)$ based on the gates
- Compute $H(x) = W(x)/G(x)$
- Prove the knowledge of exponent for each test term:
  - Raise $g^{L_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{L_{mid}(s)}$
  - Raise $g^{\alpha_L L_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{\alpha_L L_{mid}(s)}$
  - Raise $g^{R_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{R_{mid}(s)}$
  - Raise $g^{\alpha_R R_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{\alpha_R R_{mid}(s)}$
  - Raise $g^{O_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{O_{mid}(s)}$
  - Raise $g^{\alpha_O O_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{\alpha_O O_{mid}(s)}$
  - Raise $g^{\beta_L L_i(s)} g^{\beta_R R_i(s)} g^{\beta_O O_i(s)}$ to C_i for each intermediate i and multiply them, $= g^{\beta_L L_{mid}(s) + \beta_R R_{mid}(s) + \beta_O O_{mid}(s)}$
- Raise all the g^s, g^{s^2}, g^{s^3}, ... g^{s^N} to the appropriate coefficicent of $H(x)$ and multiply them to compute $g^{H(s)}$

Thus the proof consists of 8 terms (7 knowledge of exponent terms and 1 H term). The symbol π is often used for proofs. Since Veronica won't know if Philip computed all the values correctly yet, lets give them each a short hand using π. The 8 terms are:

- g^{L_π} $(= g^{L_{mid}(s)})$
- $g^{\alpha_L L_\pi}$ $(= g^{\alpha_L L_{mid}(s)})$
- g^{R_π} $(= g^{R_{mid}(s)})$
- $g^{\alpha_R R_\pi}$ $(= g^{\alpha_R R_{mid}(s)})$
- g^{O_π} $(= g^{O_{mid}(s)})$
- $g^{\alpha_O O_\pi}$ $(= g^{\alpha_O O_{mid}(s)})$
- g^{β_π} $(= g^{\beta_L L_{mid}(s) + \beta_R R_{mid}(s) + \beta_O O_{mid}(s)})$

- g^{H_π} $(= g^{H(s)})$

In brackets we include what each term should actually equal if Philip did things correctly. The short hand preserves that all the terms should be encrypted with base g, but it replaces the *mid* subscripts with π, drops the evaluation at s, and otherwise simplifies. So $g^{\alpha_L L_\pi}$ is short hand for the second of the 8 proof terms Philip computes, and it would be correct if it equals $g^{\alpha_L L_{mid}(s)}$.

8.3 Verify

Finally, Veronica can take the 8 proof terms she got from Philip and try to verify them all. For the first six terms, she does three knowledge of exponent checks using her secret values α_L, α_R, and α_O. For the next term, she does another knowledge of exponent check using some of the existing terms and using her secret values $\beta_L, \beta_R, \beta_O$. Finally, she does the divisibility check using the elliptic curve pairing. Since Philip only computes the L, R, and O polynomials in the exponent using the intermediate values, to do this final check, Veronica has to fill in the input/output values in the exponent. So in order to do the verification, she will have to compute:

$$g^{L_{io}(s)}, g^{R_{io}(s)}, g^{O_{io}(s)}$$

by taking all the $g^{L_i(s)}$ for all input/output values of i (in our example, $i = 1, 5$), raising them to the C_i, and multiplying them together,

Thus in each check she will take one or some of the terms she got from Philip (i.e. all those with a π) and combine them with values only she knows to check a result. In detail, she checks:

- **Knowledge of Exponent - Linear Combination**: The L_i, R_i, O_i are each linearly combined:
  - $(g^{L_\pi})^{\alpha_L} == g^{\alpha_L L_\pi}$
  - $(g^{R_\pi})^{\alpha_R} == g^{\alpha_R R_\pi}$
  - $(g^{O_\pi})^{\alpha_O} == g^{\alpha_O O_\pi}$
- **Knowledge of Exponent - Same Combination**: The L_i, R_i, O_i are linearly combined with the same coefficients (the C_i):
  - $(g^{L_\pi})^{\beta_L}(g^{R_\pi})^{\beta_R}(g^{O_\pi})^{\beta_O} == g^{\beta_\pi}$
- **Divisibility Check**: $W(x) = L(x)R(x) - O(x)$ and is divisible by $G(x)$:
  - $ep(g^{L_{io}(s)}g^{L_\pi}, g^{R_{io}(s)}g^{R_\pi}) == ep(g, g^{O_{io}(s)}g^{O_\pi})\,ep(g^{G(s)}, g^{H_\pi})$.

Note that in the final check she includes the *io* values she computed herself with the π values computed by Philip to get the full values. For instance if Philip computed g^{L_π} correctly as $g^{L_{mid}(s)}$ we have $g^{L_{io}(s)}g^{L_\pi} = g^{L_{io}(s)}g^{L_{mid}(s)} = g^{L_{io}(s)+L_{mid}(s)} = g^{L(s)}$.

If all of these checks are true, she successfully verified a proof! Congrats to Philip and Veronica for their success, and to you for inventing this crazy protocol for them!

But was it all worth it? The whole point of this thing was for Veronica to avoid doing too much computation. She did have to do a bunch of initial setup work, which might be costly, but only has to be done once. Once the setup is complete, the only work she has to do to verify any future instance of this computation is:

- **IO Precompute**: 6 exponents (for each of 2 input/output for each of 3 L, R, O) and 3 multiplications (for each of 3 L, R, O)
- **Knowledge of Exponent - Linear Combination**: 3 exponents
- **Knowledge of Exponent - Same Combination**: 3 exponents and 2 multiplications
- **Divisibility**: 3 pairings and 4 multiplications

Compared to just computing $y = x^5$ for herself, that's plenty more work. But notice that because of the magic of SNARKs, once she's completed the setup for a given computation, this is the only work she has to do to check that computation, *no matter how complex it is.* It could be $y = x^{10000}$, or something with 10 million operations. And in every case, she'll only have to do 12 exponents, 9 multiplications, and 3 pairings. That's incredible!

If there are more inputs and outputs (in our example there was only one of each), then she'll have to do more work for the **IO Precompute**. The amount of work scales linearly with the number of inputs and outputs. But since there's usually way more intermediate values than inputs/outputs, that's still a ton of savings for her. Nice!

There we have it folks. We've successfully boiled proving *any* number of multiplications into a carefully constructed handful of exponents, pairings, and multiplications. Wild.

As we noted in the intro, we'd start by building a SARK, a Succinct ARgument of Knowledge. We've now successfully done that. Philip proved he has knowledge of all the correct intermediate values of the computation, and he did it succinctly, with a small proof that Veronica can quickly verify. The proof is so succinct in fact that it's a constant size, no matter how big the computation is. Hard to believe, but that's what we achieved.

But there's a few issues with our design. First, each verifier (Veronica, Valerie, etc.) has to do their own setup with their own secrets, and send the results of their setup to the prover. So they have to interact directly with the prover. That makes it an interactive

proof, which isn't great. Ideally we have one single setup anyone can use, so the verifier doesn't have to interact with the prover at all. That would make it Non-interactive, a SNARK. Second, we only looked at multiplication. But what about other operations? Turns out that basically any computation can be represented as a circuit made up of multiplication and addition operations. So at the very least we'll have to take a look at addition. Third, we still have to make it "zk", by ensuring that, not only does the verifier not have to know about certain values in the computation, they actually *can't know*, even if they try to figure them out. We'll have to think about if our SNARK is zero-knowledge and if not what it takes to make it so. Finally, if we address these three points, is what we invented here really the same as the original Pinocchio zk-SNARK? Do we have to make any changes? And how does it all relate to the more modern SNARKs that have been created since? Let's take these up in the following sections.

9 Shared Setup and a Non-Interactive Proof: the SNARK

In our design, we have a single verifier, Veronica, who does the setup for herself by selecting some secret values $(s, \alpha_L, \alpha_R, \alpha_O, \beta_L, \beta_R, \beta_O)$, and computing a bunch of encryptions. Since anyone who has access to these secret values might be able to forge proofs to her, she can't share them. That means if someone else, like Valerie, also wants to verify proofs from Philip of the same computation, she has to do her own setup, with her own secrets. This is annoying. Ideally, for each circuit, we'd be able to do one shared setup for everyone.

But how can we do that if we need to keep some values a secret? We've mostly figured out how to do everything on encrypted data (in the exponent), so maybe it would be enough to have some trusted party generate the secrets, encrypt them all (raise g to the power of them), and then delete the secrets themselves. But we'd have to trust that they actually deleted the secrets, otherwise they could forge proofs! For this reason, we call this a "trusted setup".

Fortunately, cryptographers have designed so-called "multi-party computation" schemes where a group of people can get together to perform a computation on some input, without any of them being able to figure out what the input is. So we can effectively *shard* the secret across a group of parties (like Voldemort's horcruxes), do a multiparty computation to perform all the encrypted values we need for the setup, and then delete their shards. So long as at least one person is honest and deletes their piece of the secret, it's impossible to recompute the whole secret. Honestly, you could probably have invented these too, but we're trying to wrap things up here, so we won't go into detail.

So that's great! These trusted setup ceremonies have become common place in pro-

duction SNARK systems, and they're done in such a way that thousands of people can participate. You can even participate yourself if you want, and so long as you delete your part of the secret, you can be sure that no one else will be able to recover the whole secret, and thus no one will be able to forge a proof. Pretty cool stuff.

If we do this trusted setup, we have another problem. In the scheme we laid out before, for Veronica to verify the proof from Philip, she actually used the secrets directly. For instance to check the knowledge of exponent, she took the term g^{L_π} she got from Philip and raised it to her secret value α_L to check if it was equal to the $g^{\alpha_L L_\pi}$ term she got from Philip. If α_L is itself encrypted as g^{α_L} during the trusted setup, so Veronica doesn't actually know the value of α_L, then she can't do this anymore. If she wants to multiply the two values in the exponent, the only way we know how to do that is by using the pairing operation *ep*. So we'll have to modify our system a bit to do more pairing checks. And while all the core ideas are exactly the same, it will make the setup and the verification slightly more complicated. But the proof that Philip has to do is exactly the same.

There's one more tricky issue. Before, Veronica generated the secret values like s, α, β and kept them secret. But now they're going to be encrypted during the trusted setup, and the encrypted values, e.g. $g^s, g^{\alpha_L}, g^{\beta_L}$, will be public. The underlying secrets remain unknown, but is it possible for Philip to use the encrypted values for any tricky business in the proof? For instance, the α_L are involved in a knowledge of exponent check where we assume Philip can't directly multiply values by α_L. But now that we know how to do encrypted multiplication (with *ep*), and he has g^{α_L}, can't he directly multiply whatever he wants, say p, using the encrypted multiplication, $ep(g^p, g^{\alpha_L}) = j^{p\alpha_L}$? He can, but this is where we're actually lucky the *ep* multiplication only works once, since now he's stuck in base j and can't do anything with the result.

That's good, but what about the β secrets? These don't get multiplied in the exponent, they get added, and that's always easier. So it might be possible for him to use the g^{β_L} terms that are now public to somehow manipulate the g^{β_π} proof term in a way that can't be detected by the verifier. That's not good. Maybe we can introduce yet another secret value γ ("gamma") in our setup, and hide the encrypted β values even further with γ. So instead of g^{β_L}, we'll have $g^{\gamma\beta_L}$, and the verifier can do pairing multiplication with g^γ to still check things. Now even though $g^{\gamma\beta_L}$ is public, there's no way for Philip to use it because there's not supposed to be any γ in the proof so if he uses the term he'll break the verification. Awesome.

In principle, we might want to mask the encrypted α terms as well because they are also malleable in the same way. However any modification a prover makes to an α term would need to also show up in the g^{β_π} term and since including γ makes it impossible to modify the g^{β_π} term, we don't have to worry. If an α term gets modified, then we'll catch it in the β check.

Let's summarize like in the last section.

9.1 Setup

Instead of every verifier doing their own setup, now we do a single shared setup. We can think of it like having a designated trusted party who picks the random secrets, computes the setup values, and then deletes the secrets. Of course in practice we use a multi-party computation so we don't have to trust any single party.

The things we need in the setup are the same. But now we can differentiate between which things Philip needs to make a proof, and which things Veronica (or Valerie, or Victoria, etc.) need to verify a proof.

We assume the form of $G(x)$ and the $L_i(x)$, $R_i(x)$, and $O_i(x)$ have already been computed.

Here's what our trusted party has to do for the setup:

- Select a secret random value s to evaluate the polynomials at
- Select secret random values α_L, α_R, α_O for the knowledge of exponent tests
- Select secret random values β_L, β_R, β_O for the knowledge of exponent tests
- Select secret random value γ to further hide the β secrets
- Compute values necessary for proving ("proving key"):
  - Compute $g^{L_i(s)}$ and $g^{\alpha_L L_i(s)}$ for each intermediate i
  - Compute $g^{R_i(s)}$ and $g^{\alpha_R R_i(s)}$ for each intermediate i
  - Compute $g^{O_i(s)}$ and $g^{\alpha_O O_i(s)}$ for each intermediate i
  - Compute $g^{\beta_L L_i(s)} g^{\beta_R R_i(s)} g^{\beta_O O_i(s)}$ for each intermediate i
  - Compute g^s, g^{s^2}, g^{s^3}, ... g^{s^N} where N is the degree of the polynomial $H(x)$ that Philip will compute (the trusted setup participants don't know $H(x)$, but they know what degree it will be)
- Compute values necessary for verifying ("verifying key"):
  - Compute g^{α_L}, g^{α_R}, g^{α_O}
  - Compute $g^{\gamma\beta_L}$, $g^{\gamma\beta_R}$, $g^{\gamma\beta_O}$, g^{γ}
  - Compute $g^{G(s)}$
  - Compute $g^{L_i(s)}$ for each input/output i
  - Compute $g^{R_i(s)}$ for each input/output i
  - Compute $g^{O_i(s)}$ for each input/output i

- Delete all secret values s, α_L, α_R, α_O, β_L, β_R, β_O, γ

Note this is almost the same as the work Veronica had to do in her setup. The main difference is that before Veronica knew all the secret values, but now she won't, so the trusted setup has to encrypt them as well.

Notice also that we split the setup into 3 parts: the secrets, the part needed for proving, and the part needed for verifying. The secrets must be deleted after setup otherwise they can compromise everything (anyone who knows them can forge proofs). This is why they're called *toxic waste*. Once the setup is complete, we no longer need them. The parts we need for proving we call the "proving key" and the parts we need for verifying are called the "verifying key". Note these keys can be quite big. The proving key has 7 terms for each intermediate value in the computation and the verifying key has 3 terms for each input/output.

9.2 Prove

The proof is exactly the same as before. The prover gets the proving key, which contains all the same terms as in Section 8.2. The proof contains the same 8 terms as before. We label them the same way using π, and show what they should equal in brackets if Philip computed them correctly:

- g^{L_π} $(= g^{L_{mid}(s)})$
- $g^{\alpha_L L_\pi}$ $(= g^{\alpha_L L_{mid}(s)})$
- g^{R_π} $(= g^{R_{mid}(s)})$
- $g^{\alpha_R R_\pi}$ $(= g^{\alpha_R R_{mid}(s)})$
- g^{O_π} $(= g^{O_{mid}(s)})$
- $g^{\alpha_O O_\pi}$ $(= g^{\alpha_O O_{mid}(s)})$
- g^{β_π} $(= g^{\beta_L L_{mid}(s)+\beta_R R_{mid}(s)+\beta_O O_{mid}(s)})$
- g^{H_π} $(= g^{H(s)})$

9.3 Verify

Verification is similar as before, except now the verifier could be anyone, so they won't know the secret values. Instead, they'll have the verifying key, which contains:

$$g, g^{\alpha_L}, g^{\alpha_R}, g^{\alpha_O}, g^{\gamma\beta_L}, g^{\gamma\beta_R}, g^{\gamma\beta_O}, g^{\gamma}, g^{G(s)}, \{g^{L_i(s)}\}_{io}, \{g^{R_i(s)}\}_{io}, \{g^{O_i(s)}\}_{io}$$

Recall the notation $\{g^{L_i(s)}\}_{io}$ means we have a set of terms $g^{L_i(s)}$ for each input/output i (each i in IO).

Since we don't have access to the secrets directly (e.g. α_L), we can no longer do the knowledge of exponent checks just by raising terms to those secrets (e.g. we can't check $(g^{L_\pi})^{\alpha_L} = g^{\alpha_L L_\pi}$). However we do have g^{α_L}. We want to multiply α_L by L_π but they're both in the exponent. We can't use normal exponent laws for that but fortunately for us, we already found a way to do that using pairings! So we can do all the same checks as before, just using more pairing operations:

- **Knowledge of Exponent - Linear Combination**: The L_i, R_i, O_i are each linearly combined:
  - $ep(g^{L_\pi}, g^{\alpha_L}) == ep(g, g^{\alpha_L L_\pi})$
  - $ep(g^{R_\pi}, g^{\alpha_R}) == ep(g, g^{\alpha_R R_\pi})$
  - $ep(g^{O_\pi}, g^{\alpha_O}) == ep(g, g^{\alpha_O O_\pi})$
- **Knowledge of Exponent - Same Combination**: The L_i, R_i, O_i are linearly combined with the same coefficients (the C_i):
  - $ep(g^{L_\pi}, g^{\gamma\beta_L})ep(g^{R_\pi}, g^{\gamma\beta_R})ep(g^{O_\pi}, g^{\gamma\beta_O}) == ep(g^{\gamma}, g^{\beta_\pi})$
- **Divisibility Check**: $W(x) = L(x)R(x) - O(x)$ and is divisible by $G(x)$:
  - $ep(g^{L_{io}(s)}g^{L_\pi}, g^{R_{io}(s)}g^{R_\pi}) == ep(g, g^{O_{io}(s)}g^{O_\pi})ep(g^{G(s)}, g^{H_\pi})$.

Awesome! The last check is exactly the same as before, but now all the other checks (the knowledge of exponent checks) all have to use pairings. Pairings are expensive, so this is definitely more work than before, but that's the trade-off with making it so that anyone can verify a proof non-interactively.

Note also the use of the γ term. The prover shouldn't make any use of the γ term, but we had to introduce it so that the prover wouldn't be able to have access to g^{β_L} (or for any of the other βs). Now there's only $g^{\gamma\beta_L}$, which foils any attempt by the prover to manipulate the g^{β_π} term, since it doesn't include γ. γ is only used by the verifier.

So we were able to make the proof non-interactive, but we had to do a trusted setup, add another secret (γ), and use more pairings in the verification. Nonetheless, we pulled it off.

Congratulations, you successfully invented a SNARK!

10 Completing Pinocchio: Constants, Addition, Zero Knowledge, and Optimizations

By now we have figured out a complete SNARK! But there's still a few things we have to work out to complete Pinocchio.

For starters, our example computation only did multiplications. We noted earlier that any computation could be done with just multiplication and addition, so we'll have to at least look at how to support addition. We might also want to add constant input values into the computation. It turns out, addition and constants practically come for free, without really having to modify our existing protocol at all.

There's also the matter of zero-knowledge. We haven't really been thinking about privacy in our approach. We assumed our verifiers were lazy, and just wanted to verify cheaply that a big computation was done. But it's often also desired that there is some kind of privacy from the verifier, so even if they weren't lazy, they couldn't figure out anything about the intermediate values the prover computed. In other words, with a regular SNARK the verifier *doesn't have to know* about values used by the prover. But with a zk-SNARK, the verifier *can't know* about those values. It turns out, it's almost trivial to extend the proof to be zero-knowledge by having the prover generate a bit of randomness and include it in the proof.

Finally, the Pinocchio paper contains a helpful optimization around the β terms that simplifies the final non-interactive proof. Recall the β terms allow us to check that the same linear combination was used for combining all sub-polynomials. With this optimization, the three β terms in the verification key can be reduced to one, and more importantly, the four pairing operations in the β checks can be reduced to only two.

Let's take these up in turn.

10.1 Addition and Constants

The computation we did ($y = x^5$) was just multiplication. And so everything we figured out about how to encode W and G was based on that. But we didn't consider addition. Let's consider a different computation that does involve addition like $y = x^3 + x^2$. We actually showed this in Fig 1, but we reproduce it here in Fig 3. We still have four gates like before, but the last one is an addition gate. Before when we were figuring out how to encode W and G, we numbered each gate, defined $G(x)$ to be 0 at each gate by setting a root there, and defined $W(x) = L * R - O$ at each gate where L is the left input at the gate, R is the right input, and O is the output. But if it was an addition gate, it would have to look like $W(x) = L + R - O$ instead. So how do we have one definition of $W(x)$ that handles both?

Notice that the way we defined each of $L(x)$, $R(x)$, and $O(x)$, there's already a bunch

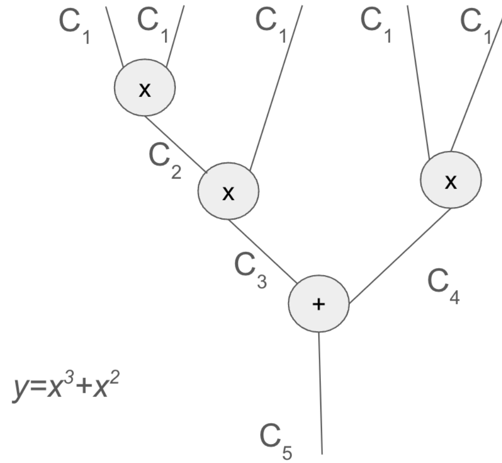

Figure 3: $y = x^3 + x^2$

of addition happening (each one is the sum of sub-polynomials multiplied by a value). Can we somehow leverage that to encode the addition gates into the structure we already have? Let's number the multiplication gates in our new example. The first gate is computing $C_2 = C_1 * C_1$. The second gate is computing $C_3 = C_2 * C_1$. The third gate is computing $C_4 = C_1 * C_1$ (this is actually the same as gate 1, but let's not worry about that yet). Finally we have our addition gate, which computes $C_5 = C_3 + C_4$.

Let's take a look at our corresponding polynomials. We know that a bunch of them are usually 0. Let's just write them out in terms of the pieces that aren't 0. For $L(x)$ we get:

$$L(x) = C_1L_1(x) + C_2L_2(x)$$

since only C_1 and C_2 are left inputs to multiplication gates (C_1 at gates 1 and 3, and C_2 at gate 2. In other words, $L(1) = C_1$, $L(3) = C_1$, and $L(2) = C_2$).

For $R(x)$ we get:

$$R(x) = C_1R_1(x)$$

since only C_1 is a right input, at all three multiplication gates. So $R(1) = C_1$, $R(2) = C_1$, and $R(3) = C_1$.

Finally for $O(x)$ we get:

$$O(x) = C_2O_2(x) + C_3O_3(x) + C_4O_4(x)$$

since C_2, C_3, and C_4 are all outputs (C_2 at gate 1, C_3 at gate 2, and C_4 at gate 3). So $O(1) = C_2$, $O(2) = C_3$, $O(3) = C_4$.

How do we include C_5 in any of this? We know that $C_5 = C_3 + C_4$. Maybe we can

rearrange for one of the other values and substitute this in? For instance, lets rearrange it as $C_3 = C_5 - C_4$ and substitute that into $O(x)$:

$$O(x) = C_2O_2(x) + (C_5 - C_4)O_3(x) + C_4O_4(x)$$

$$= C_2O_2(x) + C_5O_3(x) - C_4O_3(x) + C_4O_4(x)$$

This is kinda cool, but a bit weird, because now instead of the $C_3O_3(x)$ term we have a $C_5O_3(x)$ and a $C_4O_3(x)$. So we're still using $O_3(x)$, but C_3 is gone. We also now have a C_5, but no $O_5(x)$, and C_4 is showing up twice. Can we clean this all up? Since C_3 is an output only to gate 2, we know that $O_3(2) = 1$, but its 0 at other gates. So maybe instead of using O_3, we can encode this information into $O_4(x)$ and $O_5(x)$. For instance, instead of the $C_5O_3(x)$ term, we could use $C_5O_5(x)$, where $O_5(2) = 1$, as if we're saying that C_5 is an output of gate 2. And similarly, instead of the $-C_4O_3(x)$ term, what if we compress it into $C_4O_4(x)$? We already have $O_4(3) = 1$ since C_4 is the output of gate 3, but we could also specify that $O_4(2) = -1$, as if we're saying that C_4 is the "negative" output of gate 2. But what this really means is that the actual output of gate 2 is $C_5 - C_4$ which is exactly C_3, as we'd expect.

That's pretty interesting. We effectively managed to encode the addition gate "for free", by re-arranging the addition constraint and substituting it in for the output of one of our multiplication gates. Since the way we encode multiplication gates already includes a bunch of addition, the structure lends itself to automatically encoding addition gates. In doing so we actually got rid of C_3 and $O_3(x)$, and we didn't even need to encode the addition gate directly in $G(x)$. So the degree of our $G(x)$ and $W(x)$ polynomials actually only depend on the number of multiplication gates we have, and we can have as many addition gates as we want, for free!

This is kind of amazing. When we were working with the $y = x^5$ example we came up with a way to encode all the multiplications in polynomials $W(x)$ and $G(x)$. But it turns out we can include additions into this encoding basically for free, just by rearranging them into subtractions and allowing our sub-polynomials to equal -1 instead of just 1 and 0. That means we can easily use the same structure we used and the same proof system and everything to prove computations that include addition and multiplication, like $y = x^3 + x^2$. Cool!

The last thing we might need here is to include constants, both in multiplication and addition. For instance $y = 7x^3$ or $y = x^3 + 7$. In either case we can use the same kind of trick, encoding the 7 into the relevant sub-polynomial where ever it needs to show up. Let's start with $y = 7x^3$. If we just had $y = x^3$ we'd have two multiplication gates: $C_2 = C_1 * C_1$ and $C_3 = C_2 * C_1$. For the first gate we'd have $L_1(1) = 1$ since C_1 is a left input to gate 1. In the case of $y = 7x^3$ things are mostly the same, except we have

$C_2 = 7 * C_1 * C_1$. So we can just encode the 7 directly into $L_1(x)$ by saying $L_1(1) = 7$ since C_1 is a left input to gate 1 but we want to multiply it by 7!

That's cool. What about $y = x^3 + 7$? Here we're not multiplying by 7 but adding it. So we have $C_2 = C_1 * C_1$ for gate 1 and $C_3 = C_2 * C_1$ for gate 2 and $C_4 = C_3 + 7$ for the addition gate. If we look at the $O(x)$ we'd have $O(x) = C_2O_2(x) + C_3O_3(x)$, where $O_2(1) = 1$ and $O_3(2) = 1$. Like we did before, we can rearrange for $C_3 = C_4 - 7$ and substitute it in for C_3:

$$\begin{aligned} O(x) &= C_2O_2(x) + (C_4 - 7)O_3(x) \\ &= C_2O_2(x) + C_4O_3(x) - 7O_3(x) \end{aligned}$$

Again like before, now that the C_3 is gone, it's weird having the $O_3(x)$. We know $O_3(2) = 1$, so instead of $C_4O_3(x)$ we can do $C_4O_4(x)$ where we set $O_4(2) = 1$. But what about the $-7O_3(x)$ term? The 7 in there is just a constant, we don't have a subpolynomial for it. Can we replace $-7O_3(x)$ with a new subpolynomial that just equals -7 when $x = 2$? Since we start numbering our wires and subpolynomials at 1, why don't we introduce a special index 0 to handle these constants, like $O_0(x)$? Then we can define $O_0(2) = -7$ since at gate 2 we have to subtract this 7. We could imagine doing the same thing for left and right inputs, adding $L_0(x)$ and $R_0(x)$ to handle constants at any of the gates.

This means we have to modify our original formula for $W(x)$ slightly to add the L_0, R_0, and O_0 terms:

$$W(x) = \Big(L_0(x) + \sum_i C_iL_i(x)\Big)\Big(R_0(x) + \sum_i C_iR_i(x)\Big) - \Big(O_0(x) + \sum_i C_iO_i(x)\Big)$$

If there are constants added at any gates, these new terms will pick them up. Since the constants are known to the verifier (they're a hard-coded part of the computation), we can just include them as part of the IO terms for convenience. Not a very big change at all. But now we can really encode arbitrary circuits, with any amount of constants, additions, and multiplications. Amazing!

10.2 Zero Knowledge

We've now got a complete SNARK for arbitrary computation. The last major piece of functionality we need is to actually make it zero knowledge. In other words, we want the proof to not reveal anything about the values the prover was working with. To make sense of this a bit more, lets say Philip wants to prove that he computed $y = x^5$ for some input x and output y, without revealing either. So not only does he not reveal

the intermediate values, he doesn't want to reveal the input or output either. Instead of Veronica picking the value C_1, Philip will pick it, and not reveal it. In other words, there's now no IO values for Veronica to work with. She's just going to verify that Philip did the computation $y = x^5$ for some arbitrary value of x chosen by him.

Can we do this with the scheme we outlined above? The problem is that, with all the terms provided by Philip in the proof, he might be implicitly leaking some information about the computation. Let's suppose Philip is going to pick a number C_1 between 1 and a million to raise to the 5. Everything Philip does is a deterministic function of the publicly known values in the *proving key*, plus his one random selection of C_1. But this means Veronica could just try every possible C_1 for herself between 1 and a million, and compute the SNARK proof for herself, and see if it matches the proof sent by Philip. If so, she'll figure out Philip's C_1.

So is there some way Philip can modify his proof so that even if Veronica tries everything she still can't figure out what he picked as C_1? One idea is for Philip to generate more random values as part of the proof, and uses these to modify the proof terms. The simplest thing would be for him to generate one random value δ ("delta") and include it in all his proof terms. So for instance instead of $g^{L_{mid}(s)}$ he would compute $g^{\delta L_{mid}(s)}$. All of Veronica's checks would still be the same, but now since every term is modified by a secret δ she doesn't know, even if she tried every value of C_1, she wouldn't be able to figure out which one Philip picked.

But can we really just do this? The problem with using a single δ is the same as before when we tried to use a single β, it could accidentally leak some information because of how different terms can still be combined. So just like how the verifier had to generate three different random β values, what we really need is for the prover to generate three new random δ values: $\delta_L, \delta_R, \delta_O$. Then each of the proving terms will be mixed with a different secret and Veronica will really have no hope of figuring out what Philip picked.

So what does this look like? Recall that the final check Veronica has to do is essentially testing that $L(s)R(s) - O(s) = G(s)H(s)$. We should try to keep this the same as much as possible, but we already realized we need Philip to be able to modify the L, R, O somehow so the proof doesn't leak anything. But if he modifies the left hand side, he'll have to modify the right hand side too so they're still equal. Since $G(s)$ is fixed up front, Philip should only have to modify the $H(s)$ term on the right side. We don't know how he has to modify the $H(s)$ yet, but based on whatever he does to the L, R, O, we should be able to figure it out. Let's refer to the modified $H(s)$ as $\hat{H}(s)$ ("H-hat") for now. Ideally $\hat{H}(s)$ is some simple function of $H(s)$ itself.

Let's say we multiply each of the L, R, O by their δ. Then the left hand side of this equation becomes: $\delta_L L(s) \delta_R R(s) - \delta_O O(s)$. But we're looking for a way to express $\hat{H}(s)$ in terms of $H(s)$. We'd need to be able to factor something out so we can apply a corresponding modification to the right side. But there doesn't seem to be much we can

do here. So maybe instead of multiplying by δs we can add them:

$$(L(s) + \delta_L)(R(s) + \delta_R) - (O(s) + \delta_O)$$

If we expand and re-arrange this we'd get:

$$= L(s)R(s) - O(s) + \delta_L R(s) + \delta_R L(s) + \delta_L \delta_R - \delta_O$$

setting this equal to the right side we have:

$$L(s)R(s) - O(s) + \delta_L R(s) + \delta_R L(s) + \delta_L \delta_R - \delta_O = G(s)\hat{H}(s)$$

Now we know the $L(s)R(s) - O(s)$ term equals $G(s)H(s)$ so we can plug that in:

$$G(s)H(s) + \delta_L R(s) + \delta_R L(s) + \delta_L \delta_R - \delta_O = G(s)\hat{H}(s)$$

Now if we divide everything by $G(s)$ we can get an equation for $\hat{H}(s)$ in terms of $H(s)$!

$$H(s) + \frac{\delta_L R(s) + \delta_R L(s) + \delta_L \delta_R - \delta_O}{G(s)} = \hat{H}(s)$$

That's promising but still a bit messy. Notice that all the delta terms are in the numerator, which is divided by $G(s)$. So what if instead of just adding δ terms, we add $\delta G(s)$. Then that fraction would cancel out:

$$(L(s) + \delta_L G(s))(R(s) + \delta_R G(s)) - (O(s) + \delta_O G(s)) = G(s)\hat{H}(s)$$

Expanding:

$$L(s)R(s) - O(s) + \delta_L G(s)R(s) + \delta_R G(s)L(s) + \delta_L G(s)\delta_R G(s) - \delta_O G(s) = G(s)\hat{H}(s)$$

Substituting $G(s)H(s)$ on the left and then dividing everything by $G(s)$:

$$G(s)H(s) + \delta_L G(s)R(s) + \delta_R G(s)L(s) + \delta_L G(s)\delta_R G(s) - \delta_O G(s) = G(s)\hat{H}(s)$$

$$H(s) + \delta_L R(s) + \delta_R L(s) + \delta_L \delta_R G(s) - \delta_O = \hat{H}(s)$$

Nice! That's a pretty clean formula for $\hat{H}(s)$. Based on this we know how Philip has to modify his L, R, O, and H terms to make them zero-knowledge but still verifiable:

- $L : L(s) + \delta_L G(s)$

- $R : R(s) + \delta_R G(s)$
- $O : O(s) + \delta_O G(s)$
- $H : H(s) + \delta_L R(s) + \delta_R L(s) + \delta_L \delta_R G(s) - \delta_O$

But can he even compute all that? Remember he has to do all his math in the exponent. He doesn't have values like $L(s)$ and $H(s)$, he can only get $g^{L(s)}$ and $g^{H(s)}$. Let's start with just the L term. He does have δ_L directly (it's a secret he generated), and he computes $g^{L(s)}$. He needs to compute $g^{L(s)+\delta_L G(s)}$, but he doesn't have $G(s)$ directly. We do have $g^{G(s)}$ in the verifying key since it's needed by the verifiers, so we can just include it in the proving key as well. Then Philip can do

$$g^{L(s)}(g^{G(s)})^{\delta_L} = g^{L(s)+\delta_L G(s)}$$

to compute exactly what he needs. Sweet! He can do the same thing for R and O. But what about the α and β terms? Let's start with the α_L term. Just like with L, he can compute $g^{\alpha_L L(s)}$, but he needs $g^{\alpha_L (L(s)+\delta_L G(s))} = g^{\alpha_L L(s)+\alpha_L \delta_L G(s)}$. He doesn't have α_L, but we did have g^{α_L} in the verifying key. If we added $g^{\alpha_L G(s)}$ to the proving key for him, then he could raise it to δ_L and multiply it by $g^{\alpha_L L(s)}$ to get exactly what he needs. So that's good.

What about the β term? Recall that the β term combines the L, R, O terms, each with their own β, into one proof term that looks like $g^{\beta_L L(s)+\beta_R R(s)+\beta_O O(s)}$. And Veronica is supposed to also combine the L, R, O proof terms she gets to check them against the β proof term. But now the L, R, O terms have the extra $+\delta G(s)$ in the exponent. So when Veronica combines them all the term she gets will look like:

$$\beta_L(L(s) + \delta_L G(s)) + \beta_R(R(s) + \delta_R G(s)) + \beta_O(O(s) + \delta_O G(s))$$

Expanding and re-arranging for clarity:

$$= (\beta_L L(s) + \beta_R R(s) + \beta_O O(s)) + (\beta_L \delta_L G(s) + \beta_R \delta_R G(s) + \beta_O \delta_O G(s))$$

The first part is what we had before for the β term, and the second part with the δs is what makes it zero knowledge. So Philip needs to take that into account when producing the new β term himself. The problem again is he doesn't know $G(s)$ or the βs directly. So again we'll need to add more values to the proving key to help him out. If we add $g^{\beta_L G(s)}$ and same for R and O, then he can raise those to the respective δ and multiply everything together to get what he needs. Phew!

Interestingly, while the Pinocchio paper mentions how to make the protocol zero-knowledge, the protocol it lays out is just a SNARK, not a zk-SNARK. As if making it zk is seen as a trivial addition. So let's try to put it all together now like we did before

into a clear protocol with a setup, proving, and verifying phase, showing what all the pieces are to get a full zk-SNARK!

10.2.1 Setup

We saw above that we have to add a few more terms to the proving key so that the prover can compute everything appropriately. Namely the new terms we need are:

$$g^{G(s)}, g^{\alpha_L G(s)}, g^{\alpha_R G(s)}, g^{\alpha_O G(s)}, g^{\beta_L G(s)}, g^{\beta_R G(s)}, g^{\beta_O G(s)}$$

since he will need to raise each of these to a secret δ value he generates. By adding these to the proving key, he avoids needing to the know the $G(s)$, α, or β values directly.

Again, we're assuming here a trusted setup so we can have a proper non-interactive protocol. Except now we will also make it zero-knowledge - a true zk-SNARK. The trusted setup is the same as before except we add the new terms with $G(s)$ to the proving key:

- Select a secret random value s to evaluate the polynomials at
- Select secret random values α_L, α_R, α_O for the knowledge of exponent tests
- Select secret random values β_L, β_R, β_O for the knowledge of exponent tests
- Select secret random value γ to further hide the βs
- Compute values necessary for proving ("proving key"):
  - Compute $g^{L_i(s)}$ and $g^{\alpha_L L_i(s)}$ for each intermediate i
  - Compute $g^{R_i(s)}$ and $g^{\alpha_R R_i(s)}$ for each intermediate i
  - Compute $g^{O_i(s)}$ and $g^{\alpha_O O_i(s)}$ for each intermediate i
  - Compute $g^{\beta_L L_i(s)} g^{\beta_R R_i(s)} g^{\beta_O O_i(s)}$ for each intermediate i
  - Compute g^s, g^{s^2}, g^{s^3}, ... g^{s^N} where N is the degree of the polynomial $H(x)$ that Philip will compute (the trusted setup participants don't know $H(x)$, but they know what degree it will be)
  - Compute $g^{G(s)}$ to support zero-knowledge proving
  - Compute $g^{\alpha_L G(s)}$, $g^{\alpha_R G(s)}$, and $g^{\alpha_O G(s)}$ to support zero-knowledge proving
  - Compute $g^{\beta_L G(s)}$, $g^{\beta_R G(s)}$, $g^{\beta_O G(s)}$ to support zero-knowledge proving
- Compute values necessary for verifying ("verifying key"):
  - Compute g^{α_L}, g^{α_R}, g^{α_O}
  - Compute $g^{\gamma\beta_L}$, $g^{\gamma\beta_R}$, $g^{\gamma\beta_O}$, g^{γ}

- Compute $g^{G(s)}$
- Compute $g^{L_i(s)}$ for each input/output i and for $i = 0$
- Compute $g^{R_i(s)}$ for each input/output i and for $i = 0$
- Compute $g^{O_i(s)}$ for each input/output i and for $i = 0$

10.2.2 Prove

The proof is similar as before, except now the prover has to generate the δ values and do a bit more work so the proof is zero-knowledge. We assume they compute the base values exactly as before, so here we will only indicate how they must be extended with the δ values and the new terms provided by the proving key. Once again in brackets we include what the proof terms should like if they were computed correctly. Unfortunately they're a bit more complex than before to incldue all the δ stuff, but it's not so bad!

Recall the prover has to construct the polynomial

$$W(x) = \Big(L_0(x) + \sum_i C_i L_i(x)\Big)\Big(R_0(x) + \sum_i C_i R_i(x)\Big) - \Big(O_0(x) + \sum_i C_i O_i(x)\Big)$$

They have to divide to get $H(x) = W(x)/G(x)$. Then they can compute a zk-SNARK using the terms in the proving key:

- Select secret random values δ_L, δ_R, δ_O to make the proof zero-knowledge
- Compute $g^{L_{mid}(s)}$, $g^{R_{mid}(s)}$, $g^{O_{mid}(s)}$ as before
- Compute $g^{\alpha_L L_{mid}(s)}$, $g^{\alpha_R R_{mid}(s)}$, $g^{\alpha_O O_{mid}(s)}$ as before
- Compute $g^{\beta_L L_{mid}(s)+\beta_R R_{mid}(s)+\beta_O O_{mid}(s)}$ as before
- Extend each proof term using the new terms in the proving key and the secret δ values:
  - g^{L_π} $\big(= g^{L_{mid}(s)}\big(g^{G(s)}\big)^{\delta_L} = g^{L_{mid}(s)+\delta_L G(s)}\big)$
  - $g^{\alpha_L L_\pi}$ $\big(= g^{\alpha_L L_{mid}(s)}\big(g^{\alpha_L G(s)}\big)^{\delta_L} = g^{\alpha_L(L_{mid}(s)+\delta_L G(s))}\big)$
  - g^{R_π} $\big(= g^{R_{mid}(s)}\big(g^{G(s)}\big)^{\delta_R} = g^{R_{mid}(s)+\delta_R G(s)}\big)$
  - $g^{\alpha_R R_\pi}$ $\big(= g^{\alpha_R R_{mid}(s)}\big(g^{\alpha_R G(s)}\big)^{\delta_R} = g^{\alpha_R(R_{mid}(s)+\delta_R G(s))}\big)$
  - g^{O_π} $\big(= g^{O_{mid}(s)}\big(g^{G(s)}\big)^{\delta_O} = g^{O_{mid}(s)+\delta_O G(s)}\big)$
  - $g^{\alpha_O O_\pi}$ $\big(= g^{\alpha_O O_{mid}(s)}\big(g^{\alpha_O G(s)}\big)^{\delta_O} = g^{\alpha_O(O_{mid}(s)+\delta_O G(s))}\big)$
  - g^{β_π} $\big(= g^{\beta_L L_{mid}(s)+\beta_R R_{mid}(s)+\beta_O O_{mid}(s)}\big(g^{\beta_L G(s)}\big)^{\delta_L}\big(g^{\beta_R G(s)}\big)^{\delta_R}\big(g^{\beta_O G(s)}\big)^{\delta_O}$

$$= g^{\beta_L(L_{mid}(s)+\delta_L G(s))+\beta_R(R_{mid}(s)+\delta_R G(s))+\beta_O(O_{mid}(s)+\delta_O G(s))}\big)$$

- g^{H_π} $(= g^{H(s)}(g^{R_{mid}(s)})^{\delta_L}(g^{L_{mid}(s)})^{\delta_R}(g^{G(s)})^{\delta_L\delta_R}g^{-\delta_O}$

$$= g^{H(s)+\delta_L R_{mid}(s)+\delta_R L_{mid}(s)+\delta_L\delta_R G(s)-\delta_O})$$

Phew! Recall we constructed these terms so that all our verification checks would work exactly as before.

10.2.3 Verify

Verification is exactly the same as before, with no difference! All the complexity of making the proof zero-knowledge was hidden by the prover. The verifier isn't any the wiser. All the checks work exactly like before except now every proof term is perfectly zero-knowledge because the inclusion of the prover's secret δ values randomizes everything so that the terms don't leak any information. So the verifier can still check everything was done correctly, but without being able to glean any information at all!

- **Knowledge of Exponent - Linear Combination**: The L_i, R_i, O_i are each linearly combined:
  - $ep(g^{L_\pi}, g^{\alpha_L}) == ep(g, g^{\alpha_L L_\pi})$
  - $ep(g^{R_\pi}, g^{\alpha_R}) == ep(g, g^{\alpha_R R_\pi})$
  - $ep(g^{O_\pi}, g^{\alpha_O}) == ep(g, g^{\alpha_O O_\pi})$
- **Knowledge of Exponent - Same Combination**: The L_i, R_i, O_i are linearly combined with the same coefficients (the C_i):
  - $ep(g^{L_\pi}, g^{\beta_L\gamma})ep(g^{R_\pi}, g^{\beta_R\gamma})ep(g^{O_\pi}, g^{\beta_O\gamma}) == ep(g^\gamma, g^{\beta_\pi})$
- **Divisibility Check**: $W(x) = L(x)R(x) - O(x)$ and is divisible by $G(x)$:
  - $ep(g^{L_{io}(s)}g^{L_\pi}, g^{R_{io}(s)}g^{R_\pi}) == ep(g, g^{O_{io}(s)}g^{O_\pi})ep(g^{G(s)}, g^{H_\pi})$.

And that's that! We have finally completed the protocol for a full zk-SNARK. Congrats!

10.3 Optimization

Goodness gracious, is there really more? We could absolutely stop here, because by now, we're done. If you go and read the original Pinocchio paper, or see the protocol written up somewhere, you should now be able to recognize exactly the protocols we sketched above (depending on whether you're looking at the interactive, non-interactive, or zero-knowledge non-interactive versions). But the actual Pinocchio protocol has one more minor modification that optimizes the proof ever so slightly. It's really not a big deal,

but it reduces the number of pairings that need to be done, and its the standard form of the protocol, so we might as well cover it.

Remember our β term? We introduced three values, β_L, β_R, β_O. And in the non-interactive version of the protocol, we also had to introduce a blinding factor γ so we could safely include the encrypted values of these secrets in the public verification key. Then, verifying this term required four pairing operations that looked like this:

$$ep(g^{L_\pi}, g^{\gamma\beta_L})ep(g^{R_\pi}, g^{\gamma\beta_R})ep(g^{O_\pi}, g^{\gamma\beta_O}) == ep(g^\gamma, g^{\beta_\pi})$$

It seems like a lot of pairings just to verify one term. Can we do better somehow?

The point of having three βs instead of just one was to prevent any kind of malleability in the exponent of the g^{β_π} proof term. We needed each of the L, R, and O to be multiplied by something different. One idea then is to just always make sure we're multiplying the L, R, and O values by different random values, no matter what. For instance, what if we generated a new secret random value r_L and then everywhere we have g^L we instead do $g^{r_L L}$? Could that let us get rid of the extra β values and reduce the number of pairing operations we have to do?

Let's see. First let's look at the final divisibility check. If we replace all g^L with $g^{r_L L}$ and all g^R with $g^{r_R R}$, then the left side of that check looks like:

$$\begin{aligned} ep(g^{r_L L_{io}(s)} g^{r_L L_\pi}, g^{r_R R_{io}(s)} g^{r_R R_\pi}) &= ep(g^{r_L L(s)}, g^{r_R R(s)}) \\ &= j^{r_L r_R L(s) R(s)} \end{aligned}$$

If we also introduce r_O for the O terms then the only way we'd be able to make this work is if r_O is not random but actually $r_O = r_L r_R$. But if we do this, then actually we don't need three separate β terms anymore, we can just use one, since the random r_L and r_R take care of ensuring no malleability is possible!

Let's introduce some more notation to simplify. Let:

$$g_L = g^{r_L}; g_R = g^{r_R}; g_O = g^{r_O}; r_O = r_L r_R$$

Then for instance we can write

$$g^{r_L L_i(s)} = g_L^{L_i(s)}$$

so the r values get absorbed into the subscript of the g. The magic here is that by reducing the three β secrets to one β secret but adding two new r_L and r_R secrets and using them everywhere, we can actually reduce the number of pairing checks we have to do in the verification.

So let's lay out the final protocol now in terms of setup, proving, and verifying. We'll include the optimization we just discussed, as well as making it zk. It will be very similar

to before, with minor changes in each to accommodate this optimization.

10.3.1 Setup

The setup is the same as before, except we have the new secrets r_L and r_R and instead of the β_L, β_R, and β_O secrets we're just going to have one β secret. The trusted setup must:

- Select a secret random value s to evaluate the polynomials at
- Select secret random values r_L and r_R, set $r_O = r_L r_R$, and define $g_L = g^{r_L}$, $g_R = g^{r_R}$, $g_O = g^{r_O}$
- Select secret random values α_L, α_R, α_O for the knowledge of exponent tests
- Select secret random value β for the knowledge of exponent test
- Select secret random value γ to hide β
- Compute values necessary for proving ("proving key"):
    - Compute $g_L^{L_i(s)}$ and $g_L^{\alpha_L L_i(s)}$ for each intermediate i
    - Compute $g_R^{R_i(s)}$ and $g_R^{\alpha_R R_i(s)}$ for each intermediate i
    - Compute $g_O^{O_i(s)}$ and $g_O^{\alpha_O O_i(s)}$ for each intermediate i
    - Compute $g_L^{\beta L_i(s)} g_R^{\beta R_i(s)} g_O^{\beta O_i(s)}$ for each intermediate i
    - Compute g^s, g^{s^2}, g^{s^3}, ... g^{s^N} where N is the degree of the polynomial $H(x)$ that Philip will compute (the trusted setup participants don't know $H(x)$, but they know what degree it will be)
    - Compute $g_L^{G(s)}$, $g_R^{G(s)}$, $g_O^{G(s)}$ to support zero-knowledge proving
    - Compute $g_L^{\alpha_L G(s)}$, $g_R^{\alpha_R G(s)}$, and $g_O^{\alpha_O G(s)}$ to support zero-knowledge proving
    - Compute $g_L^{\beta G(s)}$, $g_R^{\beta G(s)}$, and $g_O^{\beta G(s)}$ to support zero-knowledge proving
- Compute values necessary for verifying ("verifying key"):
    - Compute g^{α_L}, g^{α_R}, g^{α_O}
    - Compute $g^{\gamma\beta}$, g^{γ}
    - Compute $g_O^{G(s)}$
    - Compute $g_L^{L_i(s)}$ for each input/output i
    - Compute $g_R^{R_i(s)}$ for each input/output i
    - Compute $g_O^{O_i(s)}$ for each input/output i

10.3.2 Prove

The proof is the same as before. The only thing that's changed is many of the gs in the proving key are replaced by either g_L, g_R or g_O but that's invisible to the prover. Also there's only one β. Prover does everything exactly the same, but we'll write it out for completeness, with the updated subscripts on g.

- Select secret random values δ_L, δ_R, δ_O
- Compute $g_L^{L_{mid}(s)}$, $g_R^{R_{mid}(s)}$, $g_O^{O_{mid}(s)}$ as before
- Compute $g_L^{\alpha_L L_{mid}(s)}$, $g_R^{\alpha_R R_{mid}(s)}$, $g_O^{\alpha_O O_{mid}(s)}$ as before
- Compute $g_L^{\beta L_{mid}(s)} g_R^{\beta R_{mid}(s)} g_O^{\beta O_{mid}(s)}$ as before
- Extend each proof term using the terms in the proving key and the secret δ values:
  - $g_L^{L_\pi}$ $(= g_L^{L_{mid}(s)} (g^{G(s)})^{\delta_L} = g_L^{L_{mid}(s)+\delta_L G(s)})$
  - $g_L^{\alpha_L L_\pi}$ $(= g_L^{\alpha_L L_{mid}(s)} (g^{\alpha_L G(s)})^{\delta_L} = g_L^{\alpha_L (L_{mid}(s)+\delta_L G(s))})$
  - $g_R^{R_\pi}$ $(= g_R^{R_{mid}(s)} (g^{G(s)})^{\delta_R} = g_R^{R_{mid}(s)+\delta_R G(s)})$
  - $g_R^{\alpha_R R_\pi}$ $(= g_R^{\alpha_R R_{mid}(s)} (g^{\alpha_R G(s)})^{\delta_R} = g_R^{\alpha_R (R_{mid}(s)+\delta_R G(s))})$
  - $g_O^{O_\pi}$ $(= g_O^{O_{mid}(s)} (g^{G(s)})^{\delta_O} = g_O^{O_{mid}(s)+\delta_O G(s)})$
  - $g_O^{\alpha_O O_\pi}$ $(= g_O^{\alpha_O O_{mid}(s)} (g^{\alpha_O G(s)})^{\delta_O} = g_O^{\alpha_O (O_{mid}(s)+\delta_O G(s))})$
  - g^{β_π} $(= g_L^{\beta L_{mid}(s)} g_R^{\beta R_{mid}(s)} g_O^{\beta O_{mid}(s)} (g_L^{\beta G(s)})^{\delta_L} (g_R^{\beta G(s)})^{\delta_R} (g_O^{\beta G(s)})^{\delta_O}$

$$= g_L^{\beta(L_{mid}(s)+\delta_L G(s))} g_R^{\beta(R_{mid}(s)+\delta_R G(s))} g_O^{\beta(O_{mid}(s)+\delta_O G(s))})$$

  - g^{H_π} $(= g^{H(s)} (g_R^{R_{mid}})^{\delta_L} (g_L^{L_{mid}})^{\delta_R} (g^{G(s)})^{\delta_L \delta_R} g^{-\delta_O})$

Rock on.

10.3.3 Verify

Verification is again basically the same as before, accounting for our updated bases and our single β value. The verifier checks:

- **Knowledge of Exponent - Linear Combination**: The L_i, R_i, O_i are each linearly combined:
  - $ep(g_L^{L_\pi}, g^{\alpha_L}) == ep(g, g_L^{\alpha_L L_\pi})$
  - $ep(g_R^{R_\pi}, g^{\alpha_R}) == ep(g, g_R^{\alpha_R R_\pi})$
  - $ep(g_O^{O_\pi}, g^{\alpha_O}) == ep(g, g_O^{\alpha_O O_\pi})$

- **Knowledge of Exponent - Same Combination**: The L_i, R_i, O_i are linearly combined with the same coefficients (the C_i):
  - $ep(g_L^{L_\pi} g_R^{R_\pi} g_O^{O_\pi}, g^{\beta\gamma}) == ep(g^\gamma, g^{\beta_\pi})$
- **Divisibility Check**: $W(x) = L(x)R(x) - O(x)$ and is divisible by $G(x)$:
  - $ep(g_L^{L_{io}(s)} g_L^{L_\pi}, g_R^{R_{io}(s)} g_R^{R_\pi}) == ep(g, g_O^{O_{io}(s)} g_O^{O_\pi}) ep(g_O^{G(s)}, g^{H_\pi})$.

There we have it folks, the complete Pinocchio zk-SNARK!

11 The World of Modern SNARKs

Wowee. We've come a long way but have successfully invented the first production zk-SNARK for ourselves. First we invented a SARK, then we made it non-interactive with a trusted setup to get a SNARK. Once we had the main SNARK construction, it was quite straight forward to make it zero-knowledge by introducing the δ terms. Our end result was a proof with 8 terms that could be verified with 8 pairing operations and some additional multiplications and exponentiations.

The expensive thing is the pairing checks. Our interactive SARK only had 3 pairing operations since the verifier knew the knowledge of exponent secrets, and didn't need to depend on the trusted setup and pairings to check them. But in the complete non-interactive SNARK proof there are 8 pairing operations. One open question after Pinocchio came out was whether it was possible to make a SNARK with fewer than 8 pairing ops. In 2016 Groth answered this affirmatively when he came up with a zk-SNARK based on Pinocchio but which only used 3 pairing ops. He did this by making a stronger assumption that allowed him to effectively combine all the knowledge of exponent checks into the single divisibility check. Maybe you could invent Groth16 for yourself too.

Shortly after Pinocchio came out, cryptographers were already building libraries for encoding any computation into a SNARK. To do this they needed a general way to represent circuits in code. We represented the circuit visually as a graph. Our example just used multiplication, but we said that any computation could be made up by multiplication and addition gates, and the SNARK could be constructed around single multiplication operations so that each one could support an arbitrary number of additions. The graph basically corresponds to a system of equations, where each equation has at most one multiplication. Before even trying to encode this in polynomials L_i, R_i, O_i, turns out we can represent it more simply as a set of matrices. Not everyone learns about matrices, but many do in senior years of high school and junior years of university. They're a convenient way to represent sets of linear combinations of variables.

The matrix representation of our circuit is called a *Rank 1 Constraint System (R1CS).* It's a system of equations where each equation is a *constraint* and has at most *1* multiplications (corresponding to the multiplication gates). It contains the same information as our graph representation, which is the same as what we encoded into the points that defined $L_i(x)$, $R_i(x)$, $O_i(x)$. It's just a more convenient way to represent what happens at each gate in terms of matrices **L, R, O** before encoding it all in polynomials. Just like we defined $L_i(x)$, $R_i(x)$, and $O_i(x)$ by points where the x-value was a gate and the y-value was 1 or 0, the R1CS matrix representation assigns those 1s and 0s to elements in the matrices **L, R, O**, where there's a row for every gate and a column for every wire. The values of all the wires (the C_i) are represented as the vector c that satisfies the matrix equation: $\mathbf{O}c = \mathbf{L}c\mathbf{R}c$. You don't need the matrices to invent SNARKS (the Pinocchio paper doesn't have them), but they are useful for programming the things[2]. The sub-polynomials we came up with are called a "Quadratic Arithmetic Program" (QAP), and the standard flow is to first represent the circuit as an R1CS matrix, and then convert it to a QAP.

Even though we skipped the matrix step, we've actually been thinking the whole time in terms of an R1CS encoding of the computation. Our whole design was based on arranging a series of single multiplications. This structure lent itself nicely to verifying things using pairing operations. But the R1CS model can also be quite inefficient at representing computations, especially more complex ones. We also had to do a trusted setup for each different computation (each R1CS).

In 2018 a new construction emerged called STARKs that removed the trusted setup altogether and used a different representation. STARKs are Scalable and Transparent (no trusted setup), but they have much larger proofs. They're based on the idea of "Interactive Oracle Proofs" (IOPs), where a prover commits to the execution of a specific computation and then the verifier checks random points. This makes the proof *much* bigger, like 100s or thousands of times bigger. The Pinocchio and Groth16 SNARKs that use pairings and trusted setups are extremely succinct, like a few hundred bytes, no matter how big the computation. STARKs are amazing, and unlike pairing-based systems they are quantum-secure, but succinct proofs continued to beckon.

In 2019 the Sonic protocol prompted a major revolution in SNARKs. It was still a succinct, pairing based proof, but it incorporated and generalized the ideas of IOPs from STARKs to make the trusted setup generic to *all computations of a given size*, meaning a single trusted setup could be re-used for many different circuits. This was a major breakthrough for pairing-based systems. Sonic popularized a new model for building SNARKs in a modular way by combining a "polynomial commitment scheme" with a "polynomial IOP." It also popularized a new way to represent the circuits themselves. The way you encode a computation as a circuit is called "arithmetization." R1CS was

[2]For a good overview, see https://www.rareskills.io/post/rank-1-constraint-system

the first way circuits were arithmetized and was what Pinocchio and Groth16 were based on. Instead of encoding everything in the R1CS style, as a series of single multiplications with possibly many additions, Sonic showed how the operation of each gate could be encoded directly as its own polynomial equation, openning the way to "custom gates." This means we no longer get addition "for free" (since now addition is directly encoded in polynomials as its own constraint), but we do get the major benefit of a universal trusted setup.

While Sonic used these new constructions instead of the older R1CS encoding, it still represented its circuit in terms of just multiplications, additions, and constants. But it opened the way to an explosion of innovation in SNARK design. Effectively, by mixing and matching different commitment schemes with different IOPs, you can make different kinds of SNARKs. It wasn't until Plonk later in 2019 that the true power of Sonic was realized to break through the limitations of R1CS.

Plonk built on Sonic to flesh out the idea of custom gates, giving rise to what's now called Plonkish-Arithmetization. Plonk's approach has been widely adopted in other SNARKs and has enabled massive speed ups in the proving time of more complex circuits and in the development of more expressive zkVMs. While Plonk built on Sonics conception of a polynomial commitment and IOP, it plugged in different choices from Sonic. At the same time other constructions were coming out that built on Sonic in other ways. Sonic and Plonk triggered a Cambrian explosion in SNARK development, and led to the more modern systems we have today.

By now, there is a whole zoo of SNARKs and SNARK-related techniques. There's courses on SNARKs that teach it all properly and plenty of materials to read to learn more. In this work, we hope to have demonstrated the beauty of the core construction, and to have provided a journey that in principle you could have come up with yourself.

In other words: you could have invented SNARKs!

12 Acknowledgments

Maksym Petkus wrote a paper called *Why and How zk-SNARK Works* that was very useful for explaining everything about the β terms. See that paper for a more rigorous explanation of everything we covered.