

# Lesson 12 - zkSNARK theory / PLONK

## Today's topics

- Polynomial Introduction
- Polynomial Commitment Schemes
- Proving Systems in general
- zkSNARK process
- Plonkish protocols

### Polynomial Introduction

A polynomial is an expression that can be built from constants and variables by means of addition, multiplication and exponentiation to a non-negative integer power.

e.g.  $3x^2 + 4x + 3$

Quote from Vitalik Buterin

"There are many things that are fascinating about polynomials. But here we are going to zoom in on a particular one: **polynomials are a single mathematical object that can contain an unbounded amount of information** (think of them as a list of integers and this is obvious)."

Furthermore, **a single equation between**

polynomials can represent an unbounded number of equations between numbers.

For example, consider the equation

$A(x)+B(x)=C(x)$ . If this equation is true, then it's also true that:

- $A(0)+B(0)=C(0)$
- $A(1)+B(1)=C(1)$
- $A(2)+B(2)=C(2)$
- $A(3)+B(3)=C(3)$

Adding, multiplying and dividing polynomials

We can add, multiply and divide polynomials, for examples

see

[https://en.wikipedia.org/wiki/Polynomial\\_arithmetic](https://en.wikipedia.org/wiki/Polynomial_arithmetic)

Roots

For a polynomial  $P$  of a single variable  $x$  in a field  $K$  and with coefficients in that field, the root  $r$  of  $P$  is an element of  $K$  such that  $P(r) = 0$

$B$  is said to divide another polynomial  $A$  when the latter can be written as

$$A = BC$$

with  $C$  also a polynomial, the fact that  $B$  divides  $A$  is denoted  $B|A$

If one root  $r$  of a polynomial  $P(x)$  of degree  $n$  is known then polynomial long division can be used to factor  $P(x)$  into the form

$$(x - r)(Q(x))$$

where

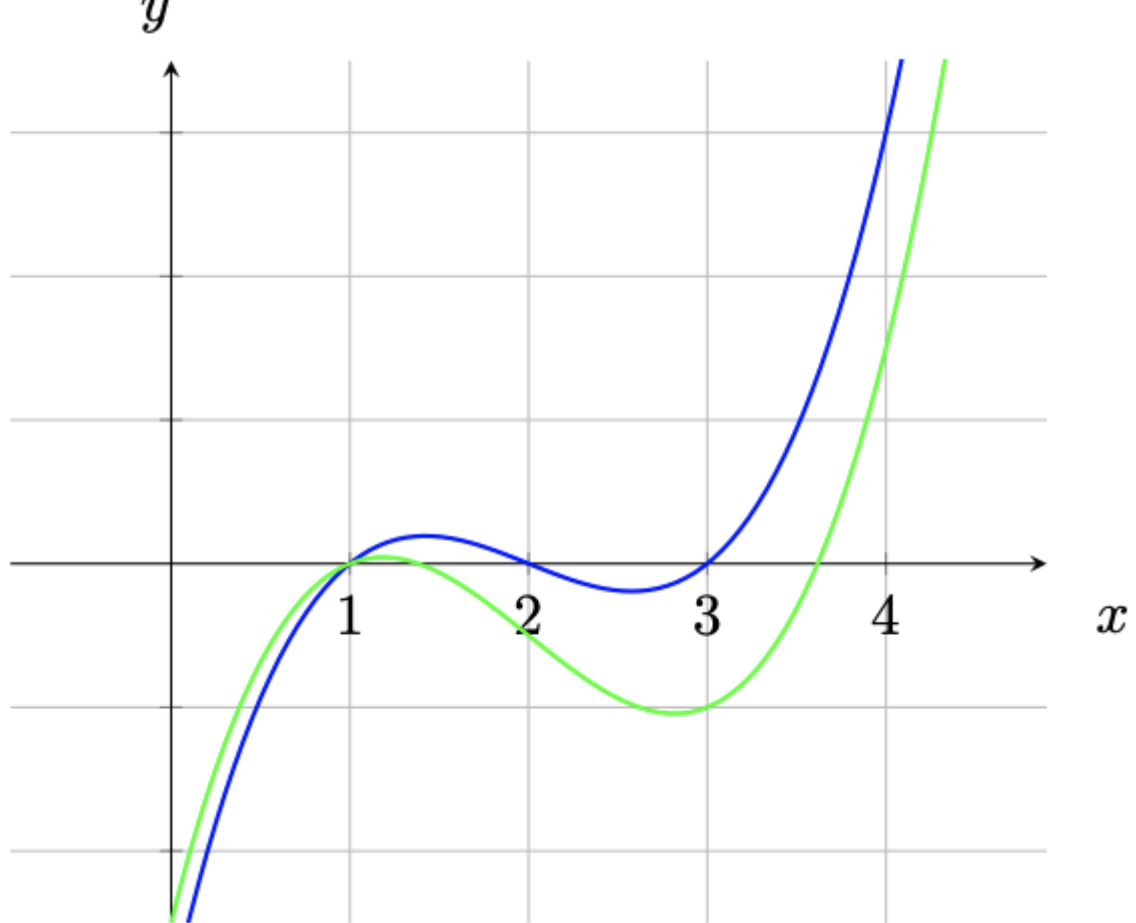
$Q(x)$  is a polynomial of degree  $n - 1$ .

$Q(x)$  is simply the quotient obtained from the division process; since  $r$  is known to be a root of  $P(x)$ , it is known that the remainder must be zero.

#### Schwartz-Zippel Lemma

"different polynomials are different at most points".

Polynomials have an advantageous property, namely, if we have two non-equal polynomials of degree at most  $d$ , they can intersect at no more than  $d$  points.



if  $f$  and  $g$  are polynomials and are equal, then  
 $f(x) = g(x)$  for all  $x$

if  $f$  and  $g$  are polynomials and are NOT equal,  
 then

$f(x) \neq g(x)$  for all pretty much any  $x$

What does it mean to say 2 polynomials are equal ?

1. They evaluate to the same value or all points
2. They have the same coefficients

If we are working with real numbers, these 2 points would go together, however that is not the case when we are working with finite fields.

For example all elements of a field of size  $q$  satisfy the identity

$$x^q = x$$

The polynomials  $X^q$  and  $X$  take the same values at all points, but do not have the same coefficients.

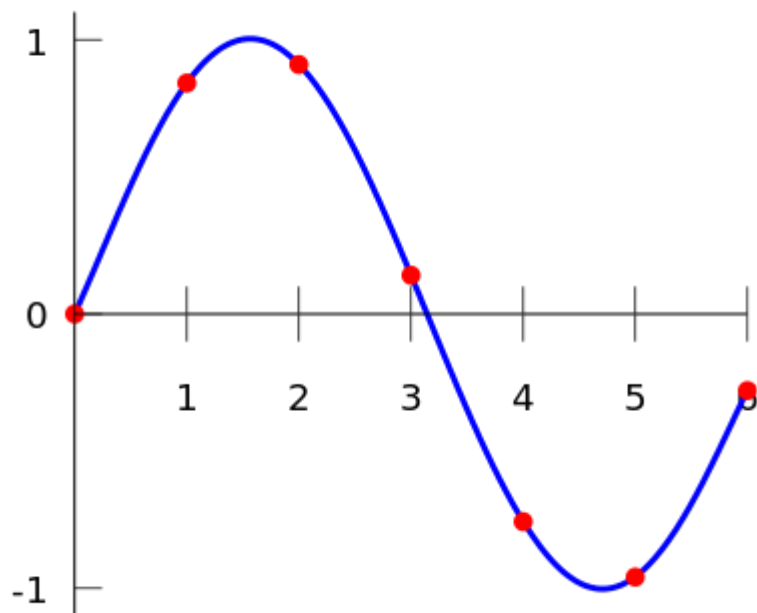
### Lagrange Interpolation

If you have a set of points then doing a Lagrange interpolation on those points gives you a polynomial that passes through all of those points.

If you have two points on a plane, you can define a single straight line that passes through both, for 3 points, a single 2nd-degree curve (e.g.  $5x^2 + 2x + 1$ ) will go through them etc.

For  $n$  points, you can create a  $n-1$  degree polynomial that will go through all of the points.

(We can use this in all sorts of interesting schemes as well as zkps)



## Representations

We effectively have 2 ways to represent polynomials

### 1. Coefficient form

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \dots$$

### 2. Point value form

$$(x_1, y_1), (x_2, y_2), \dots$$

We can switch between the two forms by evaluation or interpolation.

# Polynomial Commitment Schemes

## Introduction

A polynomial commitment is a short object that "represents" a polynomial, and allows you to verify evaluations of that polynomial, without needing to actually contain all of the data in the polynomial.

That is, if someone gives you a commitment  $c$  representing  $P(x)$ , they can give you a proof that can convince you, for some specific  $z$ , what the value of  $P(z)$  is.

There is a further mathematical result that says that, over a sufficiently big field, if certain kinds of equations (chosen before  $z$  is known) about polynomials evaluated at a random  $z$  are true, those same equations are true about the whole polynomial as well.

For example, if  $P(z) \cdot Q(z) + R(z) = S(z) + 5$  for a particular  $z$ , then we know that it's overwhelmingly likely that

$P(x) \cdot Q(x) + R(x) = S(x) + 5$  in general.

Using such polynomial commitments, we could very easily check all of the above polynomial

equations above - make the commitments, use them as input to generate  $z$ , prove what the evaluations are of each polynomial at  $z$ , and then run the equations with these evaluations instead of the original polynomials.

A general approach is to have the evaluations in a merkle tree, the leaves of which the verifier can select at random, along with merkle proof of their membership.

### Role in ZKPs

Commitment schemes generally allow the properties of

1. Binding. Given a commitment  $c$ , it is hard to compute a different pair of message and randomness whose commitment is  $c$ .

This property guarantees that there is no ambiguity in the commitment scheme, and thus after  $c$  is published it is hard to open it to a different value.

2. Hiding. It is hard to compute any information about  $m$  given  $c$ .



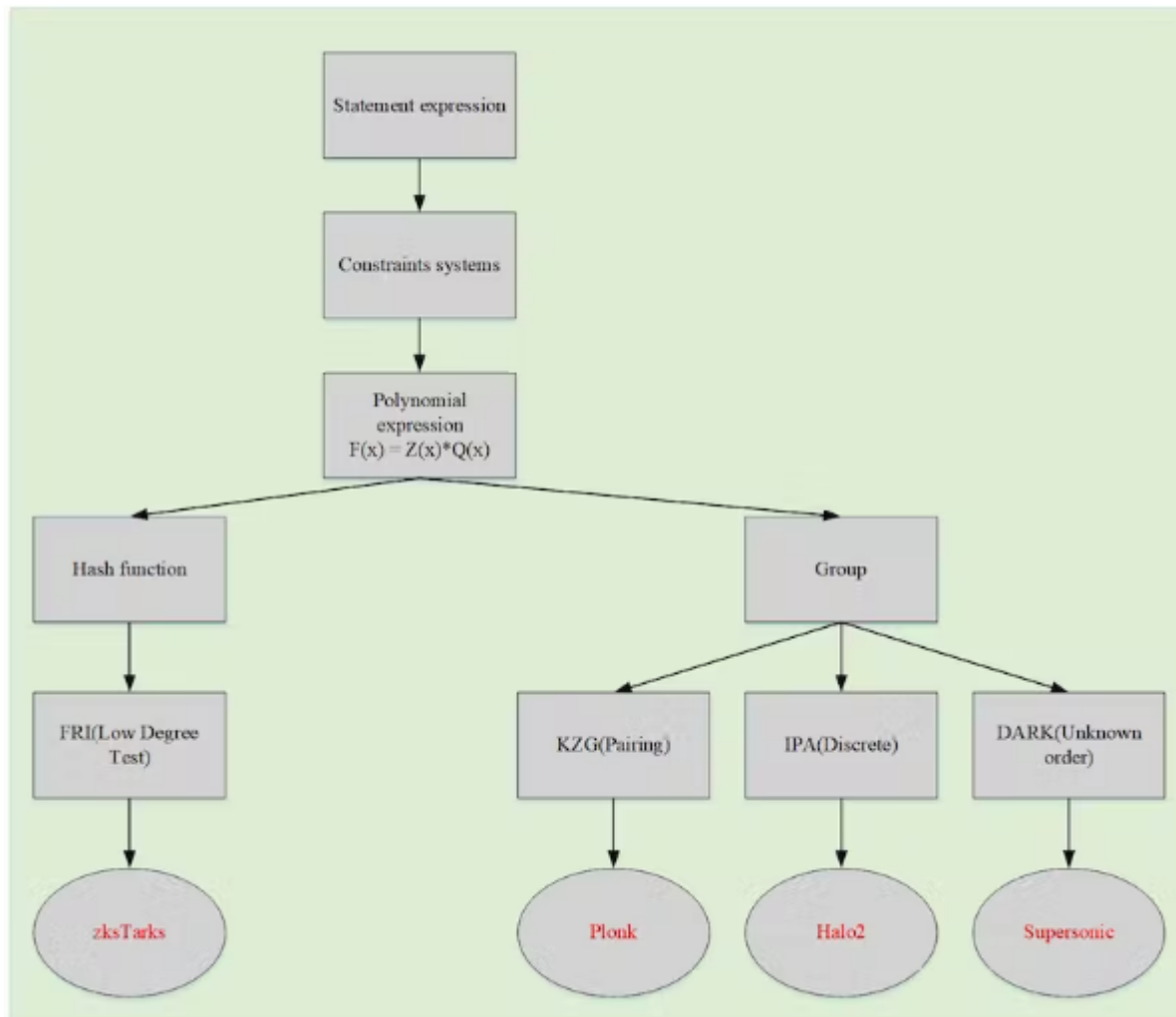
Given the size of the polynomials used in ZKPs, with say  $10^8$  terms, they help with succinctness



by reducing the size of the information that needs to be passed between the prover and verifier.



## Types of PCS



Comparison of Schemes and their underlying assumptions

Taken from [ZKP Study Group VideoSlides](#)

	hash function	pairing group	RSA group	class group	discrete log
transparent					
succinct					
unbounded					
updatable		curve-specific			
post-quantum					

	hash function	pairing group	UO group	discrete log group
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d \cdot \log(d))$	$O(d)$	$O(d \cdot \log(d))$	$O(d)$

	hash function (FRI)	pairing group (KZG)	UO group (DARK)	discrete log group (Bulletproof)
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d \cdot \log(d))$	$O(d)$	$O(d)$	$O(d)$



subexponential attacks  
(sieve based)

hash function (FRI)	pairing group (KZG)	RSA group (DARK)	class group (DARK)	discrete log (Bulletproof)
$O(1) * O(\lambda)$	$O(1) * O(\lambda^3)$	$O(1) * O(\lambda^3)$	$O(1) * O(\lambda^2)$	$O(1) * O(\lambda)$

target  
128 bits  
security

SHA256  
256 bits  
(32 bytes)

BLS12-381  
381 bits  
(48 bytes)

RSA  
3096 bits  
(387 bytes)

class group  
1827 bits  
(229 bytes)

secp256k1  
256 bits  
(32 bytes)







## Idealised proving system

There is much missed out, and assumed here, this is just to show a general process.

### Use of randomness

The prover uses randomness to achieve zero knowledge, the verifier uses randomness when generating queries to the prover, to detect cheating by the prover.

### Steps

1. Prover claims Statement  $S$
2. Verifier provides some constraints about the polynomials 
3. Prover provides (or commits to)  $P_i \dots P_k$  : polynomials 
4. Verifier provides  $z \in 0, \dots p - 1$  
5. Prover provides evaluations of polynomials:  
 $P_1(z) \dots P_k(z)$
6. Verifier decides whether to accept  $S$  

The degree expected are typically about  $10^6$  (still considered low degree)

Note the probability of accepting a false proof is  $< 10.d/p$

where  $p$  is the size of the field, so of the order of  $2^{230}$

if our finite field has  $p$  of  $\sim 2^{256}$

typically the number of queries is 3 - 10, much less than the degree

The only randomness we use here is sampling  $z$  from  $0, \dots, p-1$ , in general the randomness we use in the process is essential for both succinctness and zero knowledge

Why doesn't the verifier evaluate the polynomials themselves ?

- because, the prover doesn't actually send all the polynomials to the verifier, if they did we would lose succinctness, they contain more information than our original statement, so the prover just provides a commitment to the polynomials

What are the properties of polynomials that are important here ?

**Polynomials are good error correcting codes**

If we have polynomials of degree  $d$  over an encoding domain  $D$ , and two messages  $m_1$  and



$m_2$ , then  $m_1$  and  $m_2$  will differ at  $|D| - d$  points. This is important because we want the difference between a correct and an incorrect statement to be large, so easily found. This leads to good sampling, which helps succinctness, we need only sample a few values to be sure that the probability of error is low enough to be negligible.

### Have efficient 'batch zero testing'

This also helps with succinctness

Imagine we want to prove that a large degree polynomial  $P(x)$  (degree  $\sim 10$  million) evaluates to zero at points  $1 \dots 1$  million, but we want to do this with only one query.

Imagine that our statement is that  $P$  vanishes on these points.

If the verifier just uses sampling the prover could easily cheat by providing a point that evaluates to zero, but the other 999,999 could be non zero.

We solve this by

take a set  $S = 1 \dots 10^6$

define  $V$  as the unique polynomial that vanishes on these points

i.e.

$$(x - 1)(x - 2)(x - 3) \dots$$

the degree of  $V = \text{size of } S$

this is good because

$P(x)$  vanishes on  $S$  iff there exists  $P'(x)$  such that

1.  $P(x) = P'(x) \cdot V(x)$

2. the degree of  $P' = \text{degree of } P - \text{size of } S$

It is the introduction of  $V(x)$  that allows us to check across the whole domain

## Have "multiplication" property

We can 'wrap' a constraint around a polynomial

For example if we have the constraint  $C$ , that our evaluation will always be a zero or a one, we could write this constraint as

$$C(x) = x \cdot (x - 1) = 0$$

You could imagine this constraining an output to be a boolean, something that may be useful for computational integrity.

But here instead of  $x$  being just a point it could be the evaluation of a polynomial  $P_1(x)$  at a point

i.e.

$$C(P_1(x)) = P_1(x) \cdot (P_1(x) - 1) = 0$$

and the degrees of the polynomials produced by the multiplication then are additive so degree of  $C(x) = 2$ . degree of  $P_1(x)$

We can then make the claim, that if  $P_1(x)$  does indeed obey this constraint for our set  $S$  then as we did above we can say that there is some polynomial  $P'(x)$  such that

$$C(P_1(x)) = P'(x) \cdot V(x)$$

If  $P_1(x)$  didn't obey the constraint (for example if for one value of  $x$ ,  $P_1(x) = 93$ ) then we wouldn't be able to find such polynomials, the equality wouldn't hold and there would effectively be a remainder in the preceding equation.

---



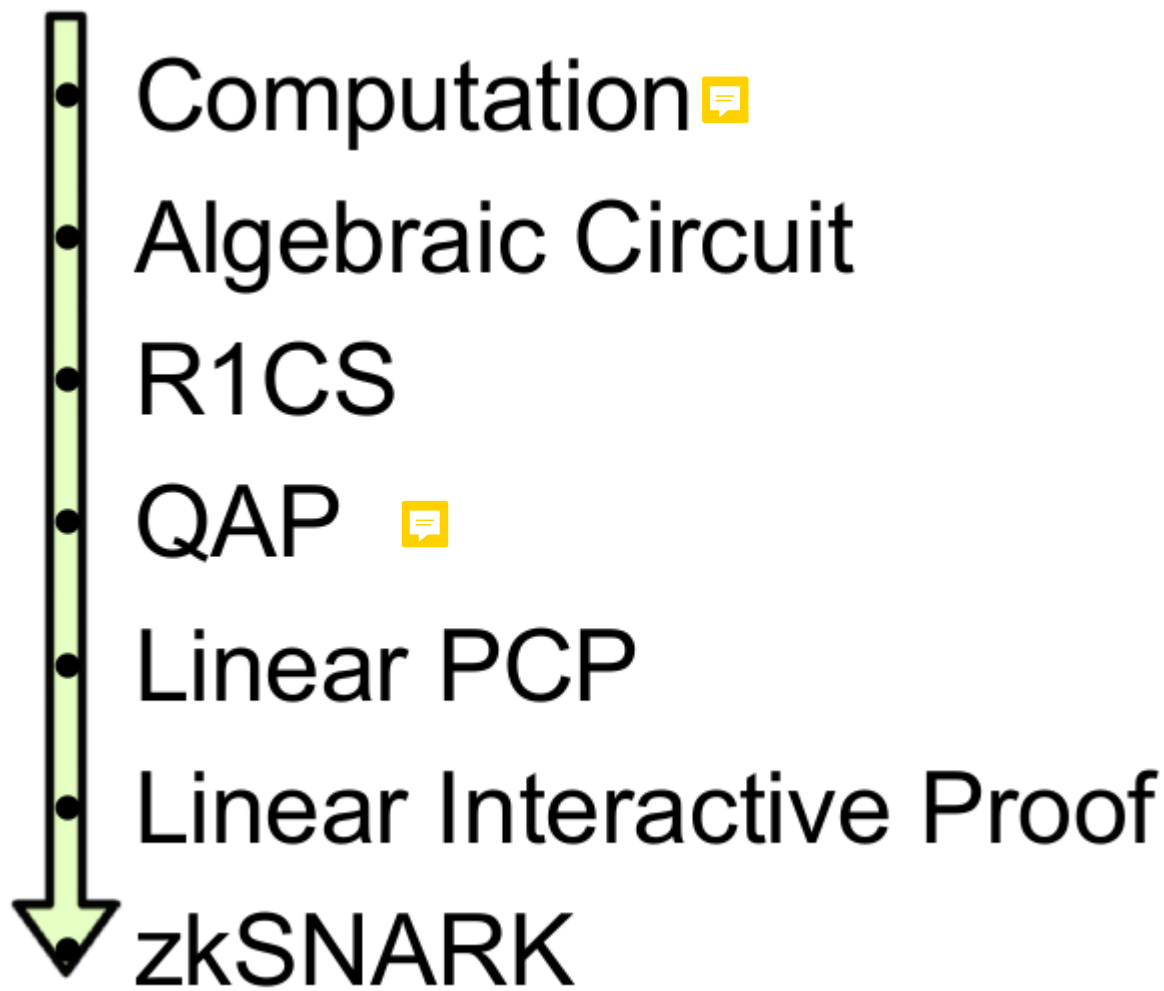
# zkSNARK Process

## General Process

- Arithmetisation
  - Flatten code
  - Arithmetic Circuit
- Polynomials
- Polynomial Commitment Scheme
- Cryptographic proving system
- Make non interactive

## Transformations in SNARKS

A diagram showing the transformations for early versions of SNARKS



## 1. Trusted Setup

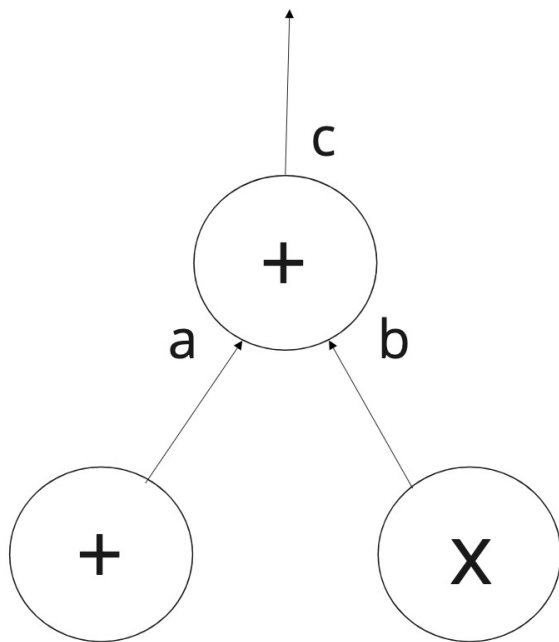
ZKSNarks require a one off set up step to produce prover and verifier keys. This step is generally seen as a drawback to zkSNARKS, it requires an amount of trust, if details of the setup are later leaked it would be possible to create false proofs.

## 2. A High Level description is turned into an arithmetic circuit

The creator of the zkSNARK uses a high level language to specify the algorithm that constitutes and tests the proof.

This high level specification is compiled into an arithmetic circuit.

An arithmetic circuit can be thought of as similar to a physical electrical circuit consisting of logical gates and wires. This circuit constrains the allowed inputs that will lead to a correct proof.



### 3. Further Mathematical refinement

The circuit is then turned into a an R1CS, and then a series of formulae called a Quadratic Arithmetic Program (QAP).

The QAP is then further refined to ensure the privacy aspect of the process. 🗨️

The end result is a proof in the form of series of bytes that is given to the verifier. The verifier can pass this proof through a verifier function to receive a true or false result.

There is no information in the proof that the verifier can use to learn any further

information about the prover or their witness.

From ZCash explanation :

"SNARKs require something called "the public parameters". The SNARK public parameters are numbers with a specific cryptographic structure that are known to all of the participants in the system. They are baked into the protocol and the software from the beginning.

The obvious way to construct SNARK public parameters is just to have someone generate a public/private keypair, similar to an ECDSA keypair, (See ZCash [explanation](#)) and then destroy the private key.

The problem is that private key. Anybody who gets a copy of it can use it to counterfeit money. (However, it cannot violate any user's privacy — the privacy of transactions is not at risk from this.)"

ZCash used a *secure multiparty computation* in which multiple people each generate a "shard" of the public/private keypair, then they each destroy their shard of the toxic waste private key, and then they all bring together their

shards of the public key to form the SNARK public parameters.

If that process works — i.e. if *at least one of the participants* successfully destroys their private key shard — then the toxic waste byproduct never comes into existence at all.

Lets look first at transforming the problem into a QAP, there are 3 steps :

- code flattening,
- creation of an arithmetic circuit
- conversion to a rank-1 constraint system (R1CS)

### Code Flattening

We are aiming to create arithmetic and / or boolean circuits from our code, so we change the high level language into a sequence of statements that are of two forms

$x = y$  (where  $y$  can be a variable or a number)  
and

$x = y \text{ (op) } z$

(where  $\text{op}$  can be  $+$ ,  $-$ ,  $*$ ,  $/$  and  $y$  and  $z$  can be variables, numbers or themselves sub-expressions).

For example we go from

```
def qeval(x):  
    y = x**3
```



```
return x + y + 5
```

to

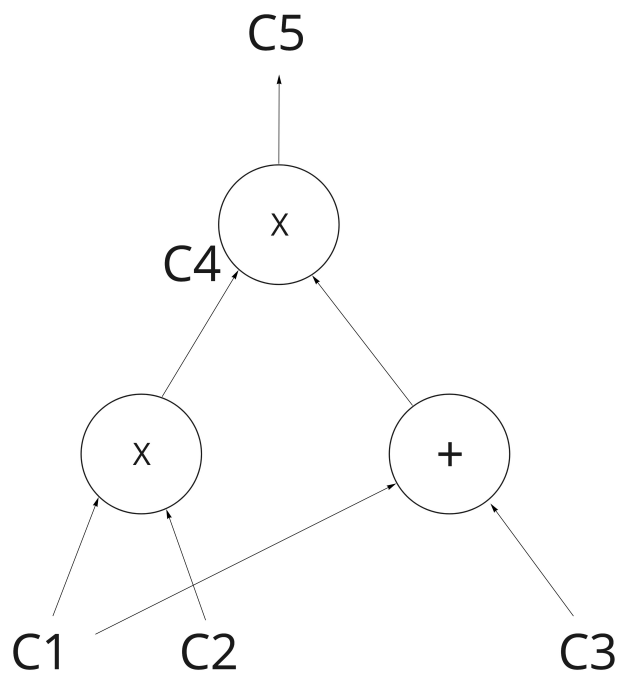
```
sym_1 = x * x
```

```
y = sym_1 * x
```


```
sym_2 = y + x
```

```
~out = sym_2 + 5
```

This is a collection of multiplication and addition gates



### Rank 1 Constraint Systems

Constraint languages can be viewed as a generalization of  functional languages:

- everything is referentially transparent and side-effect free
- there is no ordering of constraints
- composing two R1CS programs just means that their constraints are simultaneously satisfied.

(From <http://coders-errand.com/constraint-systems-for-zk-snarks/>)

The important thing to understand is that a R1CS is not a computer program, you are not asking it to produce a value from certain inputs. Instead, a R1CS is more of a verifier, it shows that an already complete computation is correct .

The arithmetic circuit is a composition of multiplicative sub-circuits (a single multiplication gate and multiple addition gates)

A rank 1 constraint system is a set of these sub-circuits expressed as constraints, each of the form:

$$AXB = C$$

where  $A, B, C$  are each linear combinations  $c_1 \cdot v_1 + c_2 \cdot v_2 + \dots$

The  $c_i$  are constant field elements, and the  $v_i$  are instance or witness variables (or 1).

- $AXB = C$  doesn't mean  $C$  is computed from  $A$  and  $B$  just that  $A, B, C$  are consistent.

More generally, an implementation of  $x = f(a, b)$  doesn't mean that  $x$  is computed from  $a$  and  $b$ , just that  $x$ ,  $a$ , and  $b$  are consistent.



Thus our R1CS contains :

- the constant 1
- all public inputs
- outputs of the function
- private inputs
- auxilliary variables

The R1CS has

- one constraint per gate;
- one constraint per circuit output.

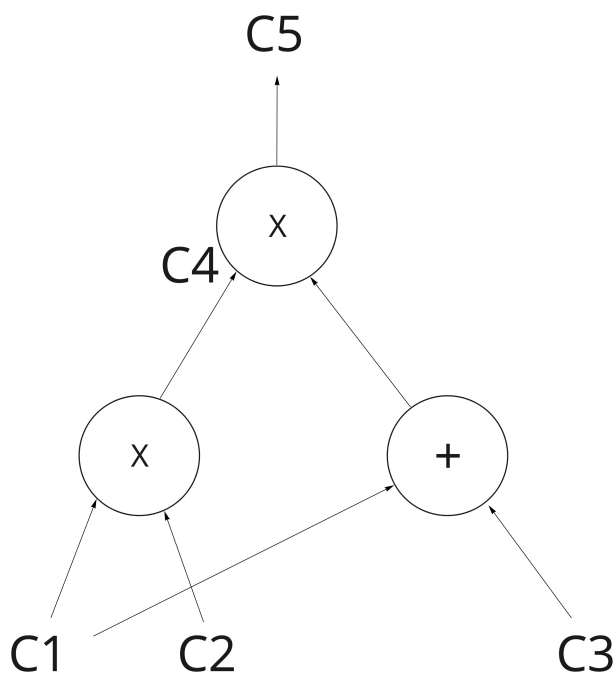
### Example

Assume Peggy wants to prove to Victor that she knows

$c_1, c_2, c_3$  such that

$$(c_1 \cdot c_2) \cdot (c_1 + c_3) = 7$$

We transform the expression above into an arithmetic circuit as depicted below



A legal assignment for the circuit is of the form:

$(c_1, \dots, c_5)$ , where  $c_4 = c_1 \cdot c_2$  and  $c_5 = c_4 \cdot (c_1 + c_3)$ .

## SNARK Process continued

### From R1CS to QAP



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

To create the polynomials we can use interpolation of the values in our R1CS

Then instead of checking the constraints in the R1CS individually, we can now check *all of the constraints at the same time* by doing the dot product check *on the polynomials*.

Because in this case the dot product check is a series of additions and multiplications of polynomials, the result is itself going to be a polynomial. If the resulting polynomial, evaluated at every x coordinate that we used above to represent a logic gate, is equal to zero, then that means that all of the checks pass; if the resulting polynomial evaluated at at least one of the x coordinate representing a logic gate gives a nonzero value, then that means

that the values going into and out of that logic gate are inconsistent

How having polynomials helps us

We can change the problem into that of knowing a polynomial with certain properties

This [paper](#) gives a reasonable explanation of how the polynomials are used to prevent the prover 'cheating'

We converted a set of vectors into polynomials that generate them when evaluated at certain fixed points.

We used these fixed points to generate a vanishing polynomial that divides any polynomial that evaluates to 0 at least on all those points.

We created a new polynomial that summarizes all constraints and a particular assignment, and the consequence is that we can verify all constraints at once if we can divide that polynomial by the vanishing one without remainder.

This division is complicated, but there are

methods (the Fast Fourier Transform) that can perform it efficiently.

---



From our QAP we have

$$L := \sum_{i=1}^m c_i \cdot Li, R := \sum_{i=1}^m c_i \cdot Ri, O := \sum_{i=1}^m c_i \cdot Oi$$

and we define the polynomial  $P$

$$P := L \cdot R - O$$

Defining the target polynomial

$$V(x) := (x - 1) \cdot (x - 2) \dots,$$

This will be zero at the points that correspond to our gates, but the  $P$  polynomial, having all the constraints information would be a some multiple of this if

- it is also zero at those points
- to be zero at those points,  $L \cdot R - O$  must equate to zero, which will only happen if our constraints are met.

So we want  $V$  to divide  $P$  with no remainder, which would show that  $P$  is indeed zero at the points.

If Peggy has a satisfying assignment it means that, defining  $L, R, O, P$  as above, there exists a polynomial  $P'$  such that

$$P = P' \cdot V$$

In particular, for any  $z \in \mathbb{F}_p$  we have

$$P(z) = P'(z) \cdot V(z)$$

Suppose now that Peggy doesn't have a satisfying witness, but she still constructs  $L, R, O, P$  as above from some unsatisfying assignment  $(c_1, \dots, c_m)(c_1, \dots, c_m)$ .

Then we are guaranteed that  $V$  does not divide  $P$ .

This means that for any polynomial  $V$  of degree at most  $d - 2$ ,  $P$  and  $L, R, O, V$  will be different polynomials.

Note that  $P$  here is of degree at most  $2(d - 1)$ ,  $L, R, O$  here are of degree at most  $d - 1$  and  $V$  here is degree at most  $d - 2$ .

Remember the Schwartz-Zippel Lemma tells us that two different polynomials of degree at most  $d$  can agree on at most  $d$  points.

---

If  $E(x)$  is a function with the following properties

- Given  $E(x)$  it is hard to find  $x$
- Different inputs lead to different outputs so if  $x \neq y$   $E(x) \neq E(y)$
- We can compute  $E(x + y)$  given  $E(x)$  and  $E(y)$

The group  $\mathbb{Z}_p^*$  with operations addition and multiplication allows this.

Here's a toy example of why Homomorphic Hiding is useful for Zero-Knowledge proofs: Suppose Alice wants to prove to Bob she knows numbers  $x, y$  such that  $x + y = 7$

1. Alice sends  $E(x)$  and  $E(y)$  to Bob.
2. Bob computes  $E(x + y)$  from these values (which he is able to do since  $E$  is an HH).
3. Bob also computes  $E(7)$ , and now checks whether  $E(x + y) = E(7)$ . He accepts Alice's proof only if equality holds.

As different inputs are mapped by  $E$  to different hidings, Bob indeed accepts the proof only if Alice sent hidings of  $x, y$  such that  $x + y = 7$ . On

the other hand, Bob does not learn  $x$  and  $y$  as he just has access to their hidings.

### Blind evaluation of a polynomial using Homomorphic Hiding

Suppose Peggy has a polynomial  $P$  of degree  $d$ , and Victor has a point  $z \in \mathbb{F}_p$  that he chose randomly.

Victor wishes to learn  $E(P(z))$ , i.e., the Homomorphic Hiding of the evaluation of  $P$  at  $z$ . Two simple ways to do this are:

1. Peggy sends  $P$  to Victor, and he computes  $E(P(z))$  by himself.
2. Victor sends  $z$  to Peggy; she computes  $E(P(z))$  and sends it to Victor.

However, in the blind evaluation problem we want Victor to learn  $E(P(z))$  without learning  $P$  which precludes the first option; and, most importantly, we don't want Peggy to learn  $z$ , which rules out the second.

Using homomorphic hiding, we can perform blind evaluation as follows.

1. Victor sends to Peggy the hidings  $E(1), E(z_1), \dots, E(z_d)$
2. Peggy computes  $E(P(z))$  from the elements sent in the first step, and sends  $E(P(z))$  to Victor. (Peggy can do this since  $E$  supports linear combinations, and  $P(z)$  is a linear combination of  $1, z_1, \dots, z_d$ )

Note that, as only hidings were sent, neither Peggy learned  $z$  nor Victor learned  $P$

The rough intuition is that the verifier has a "correct" polynomial in mind, and wishes to check the prover knows it. Making the prover blindly evaluate their polynomial at a random point not known to them, ensures the prover will give the wrong answer with high probability if their polynomial is not the correct one (Schwartz-Zippel Lemma ).

However

The fact that Peggy is able to compute  $E(P(z))$  does not guarantee she will indeed send  $E(P(z))$  to Victor, rather than some completely unrelated value.

Our process then becomes

1. Peggy chooses polynomials  $L, R, O, P, P'$
2. Victor chooses a random point  $z \in \mathbb{F}_p$ , and computes  $E(P(z))$
3. Peggy sends Victor the hidings of all these polynomials evaluated at  $z$ , i.e.  
 $E(L(z)), E(R(z)), E(O(z)), E(P(z)), E(P'(z))$

Furthermore we use

- Random values added to our  $z$  to conceal the  $z$  value
- The Knowledge of Coefficient Assumption to prove Peggy can produce a linear combination of the polynomials.

If Peggy does not have a satisfying assignment, she will end up using polynomials where the equation does not hold identically, and thus does not hold at most choices of  $z$ . Therefore, Victor will reject with high probability over his choice of  $z$ .

We now need to make our proof non interactive, for this we use the Common Reference String from the trusted setup

## Non-interactive proofs in the common reference string model

In the CRS model, before any proofs are constructed, there is a setup phase where a string is constructed according to a certain randomised process and broadcast to all parties. This string is called the CRS and is then used to help construct and verify proofs. The assumption is that the randomness used in the creation of the CRS is not known to any party – as knowledge of this randomness might enable constructing proofs of false claims.

### Why do we need this randomness

Victor is sending challenges to Peggy, if Peggy could know what exactly the challenge is going to be, she could choose its randomness in such a way that it could satisfy the challenge, even if she did not know the correct solution for the instance (that is, faking the proof).

So, Victor must only issue the challenge after Peggy has already fixed her randomness. This is why Peggy first *commits* to her randomness, and implicitly reveals it only after the challenge,

when she uses that value to compute the proof.  
That ensures two things:

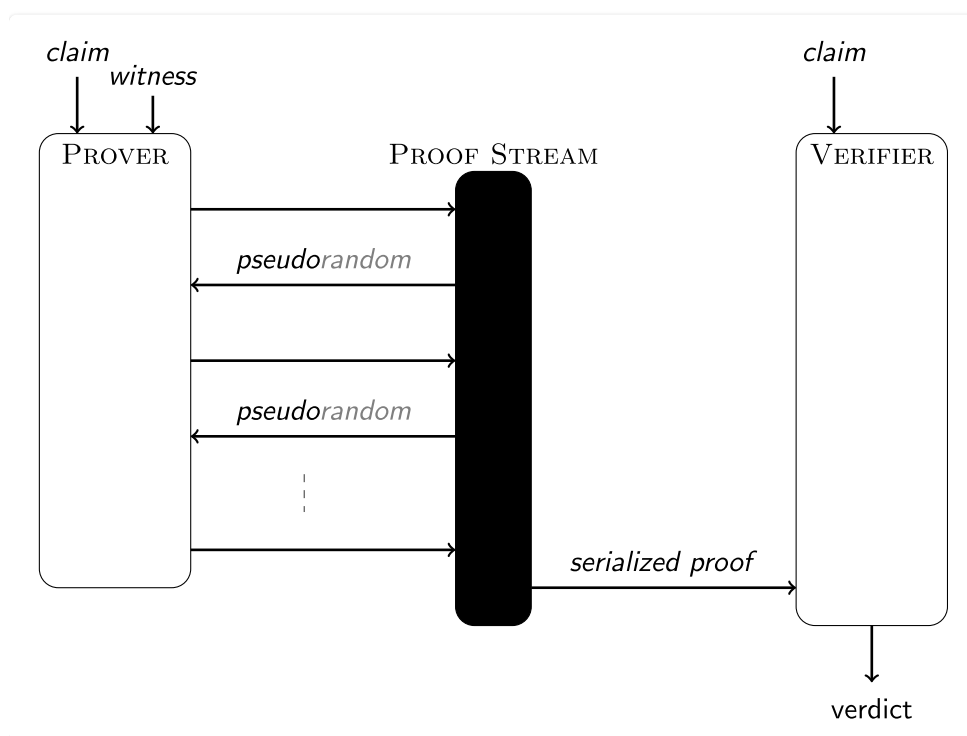
1. Victor cannot *guess* what value Peggy committed to;
  2. Peggy cannot *change* the value she committed to.
-



See <https://aszeplieniec.github.io/stark-anatomy/basic-tools>

This is a process by which we can make an interactive proof non-interactive.

It works by providing commitments to the messages that would form the interaction. The hash functions are used as a source of randomness.



## Resources

[Quadratic Arithmetic Programs: from Zero to Hero](#)

[How Plonk Works](#)

## Plonkish protocols

( fflonk, turbo PLONK, ultra PLONK, plonkup, and recently plonky2.)



### Before PLONK

Early SNARK implementations such as Groth16 depend on a common reference string, this is a large set of points on an elliptic curve.

Whilst these numbers are created out of randomness, internally the numbers in this list have strong algebraic relationships to one another. These relationships are used as short-cuts for the complex mathematics required to create proofs.

Knowledge of the randomness could give an attacker the ability to create false proofs.

A trusted-setup procedure generates a set of elliptic curve points  $G, G \cdot s, G \cdot s^2, \dots, G \cdot s^n$ , as well as  $G^2 \cdot s$ , where  $G$  and  $G^2$  are the generators of two elliptic curve groups and  $s$  is a secret that is forgotten once the procedure is finished (note that there is a multi-party version of this setup, which is secure as long as at least one of the participants forgets their share of the secret).

(The Aztec reference string goes up to the 10066396th power)

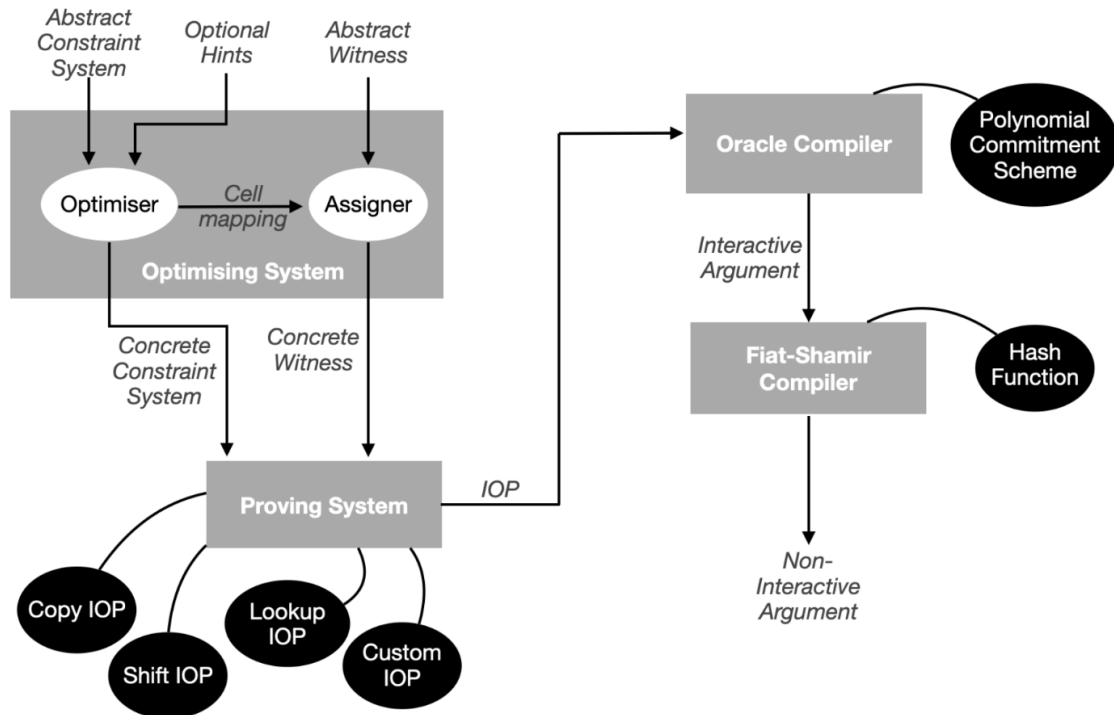
A problem remains that if you change your program and introduce a new circuit you require a fresh trusted setup.

In January 2019 Mary Maller, Sean Bowe et al released SONIC that has a universal setup, with just one setup, it could validate any conceivable circuit (up to a predefined level of complexity). This was unfortunately not very efficient, PLONK managed to optimise the process to

make the proof process feasible.

2 <sup>17</sup> Gates	PLONK		Marlin
Curve	BN254	BLS12-381 (est.)	BLS12-381
Prover Time	2.83s	4.25s	c. 30s
Verifier Time	1.4ms	2.8ms	8.5ms

A zero-knowledge proof typically consists of the following components:



### Trusted Setup

This is still needed, but it is a "universal and updateable" trusted setup.

- There is one single trusted setup for the whole scheme after which you can use the scheme with any program (up to some maximum size chosen when making the setup).
- There is a way for multiple parties to participate in the trusted setup such that it is secure as long as any one of them is honest, and this multi-party procedure is fully sequential:

