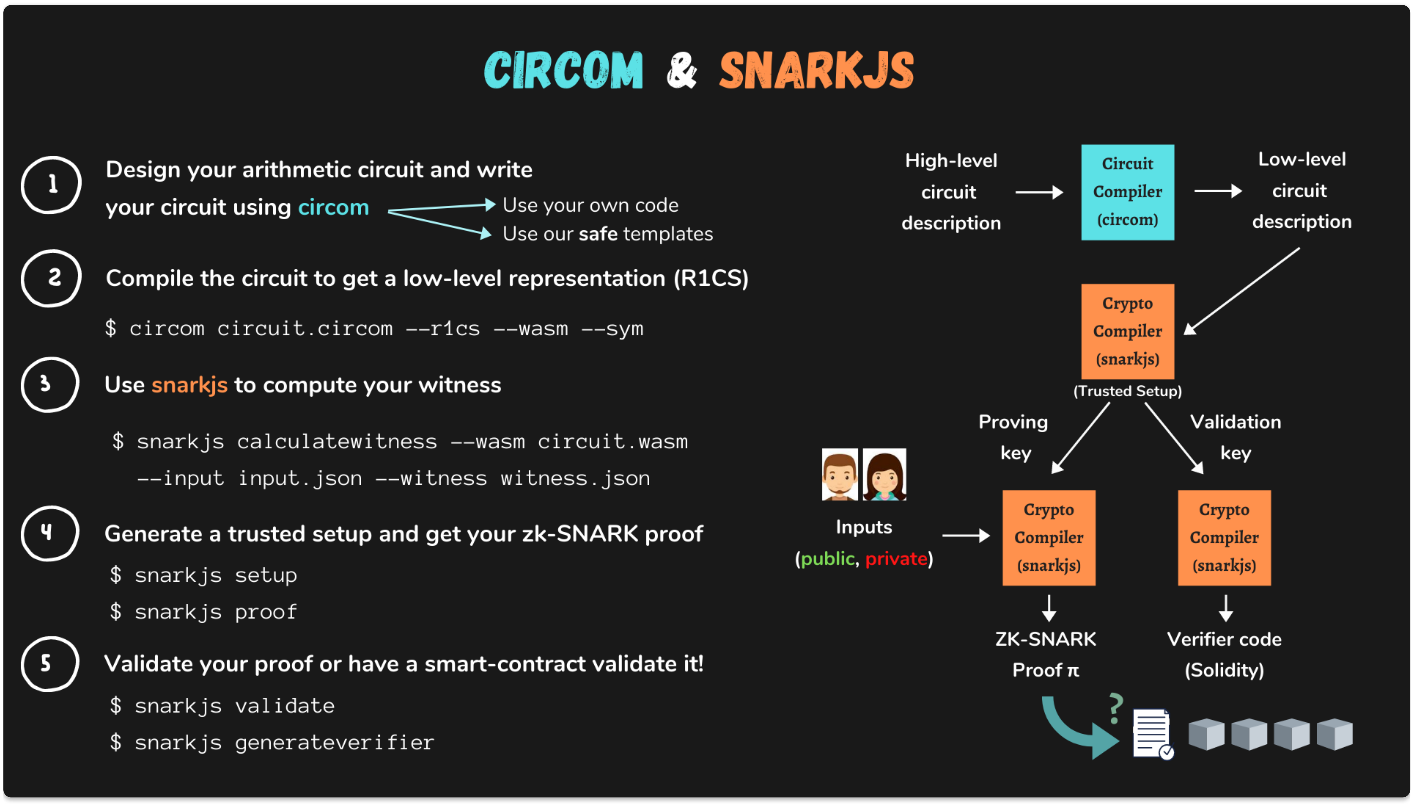
Lesson 12 - Circom / SNARK theory

Circom

See Documentation



CIRCOM & SNARKJS

1. Design your arithmetic circuit and write your circuit using circom
   - Use your own code
   - Use our **safe** templates
2. Compile the circuit to get a low-level representation (R1CS)

```
$ circom circuit.circom --r1cs --wasm --sym
```

3. Use snarkjs to compute your witness

```
$ snarkjs calculatewitness --wasm circuit.wasm
  --input input.json --witness witness.json
```

4. Generate a trusted setup and get your zk-SNARK proof

```
$ snarkjs setup
$ snarkjs proof
```

5. Validate your proof or have a smart-contract validate it!

```
$ snarkjs validate
$ snarkjs generateverifier
```

High-level circuit description
Circuit Compiler (circom)
Low-level circuit description
Crypto Compiler (snarkjs)
(Trusted Setup)
Proving key
Validation key
Inputs (public, private)
Crypto Compiler (snarkjs)
Crypto Compiler (snarkjs)
ZK-SNARK Proof π
Verifier code (Solidity)
?

Installation

Dependencies

- Rust - use rustup

  `curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh`
- Circom

  `git clone https://github.com/iden3/circom.git`

  `cd circom`

  `cargo build --release`

  `cargo install --path circom`
- Snarkjs

  `npm install -g snarkjs`

Circom Lib

SeeRepo

CircomLib

Description

- This repository contains a library of circuit templates.
- All files are copyrighted under 2018 0KIMS association and part of the free software circom (Zero Knowledge Circuit Compiler).
- You can read more about the circom language in the circom documentation webpage.

Organisation

This respository contains 5 folders:

- `circuits` : it contains the implementation of different cryptographic primitives in circom language.
- `calcpedersenbases` : set of functions in JavaScript used to find a set of points in Baby Jubjub elliptic curve that serve as basis for the Pedersen Hash.
- `doc` : it contains some circuit schemes in ASCII (must be opened with Monodraw, an ASCII art editor for Mac).
- `src` : it contains similar implementation of circuits in JavaScript.
- `test` : tests.

A description of the specific circuit templates for the `circuit` folder will be soon updated.

Circom coding

Writing circuits
Taken from the documentation

Circom allows programmers to define the constraints that define the arithmetic circuit. All constraints must be of the form `A*B + C = 0`, where `A`, `B` and `C` are linear combinations of signals.

You can define constraints in this way

```
pragma circom 2.0.0;

/*This circuit template checks that c is the multiplication of a and b.*/

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal output c;

   // Constraints.
   c <== a * b;
}
```

Signals

The arithmetic circuits built using circom operate on signals
Signals can be named with an identifier or can be stored in arrays and declared using the keyword signal.
Signals can be defined as input or output, and are considered intermediate signals otherwise.
Signals are by default private.
The programmer can distinguish between public and private signals only when defining the main component, by providing the list of public input signals.

```
  pragma circom 2.0.0;

template Multiplier2(){
   //Declaration of signals
   signal input in1;
   signal input in2;
   signal output out;
   out <== in1 * in2;
}

component main {public [in1,in2]} = Multiplier2();
```

Circom data types

- Field Element
  Integers mod the max field value, these are the default type.
- Arrays
  These hold items of the same type

```
var x[3] = [2,8,4];
var z[n+1];  // where n is a parameter of a template
var dbl[16][2] = base;
var y[5] = someFunction(n);
```

Templates and components

The mechanism to create generic circuits in Circom is the so-called templates.

They are normally parametric on some values that must be instantiated when the template is used.
The instantiation of a template is a new circuit object, which can be used to compose other circuits, so as part of larger circuits.

```
template tempid ( param_1, ... , param_n ) {
 signal input a;
 signal output b;

 .....

}
```

The instantiation of a template is made using the keyword component and by providing the necessary parameters.

`component c = tempid(v1,...,vn);`
The values of the parameters should be known constants at compile time.

Components

A component defines an arithmetic circuit has input signals, output signals and intermediate signals, and can have a set of constraints.
Components are immutable once instantiated.

```
template A(N){
   signal input in;
   signal output out;
   out <== in;
}

template C(N){
   signal output out;
   out <== N;
}
template B(N){
  signal output out;
  component a;
  if(N > 0){
     a = A(N);
  }
  else{
     a = A(0);
  }
}

component main = B(1);
```

We can create arrays of components.

```
 template MultiAND(n) {
   signal input in[n];
   signal output out;
   component and;
   component ands[2];
   var i;
```

```
    if (n==1) {
        out <== in[0];
    } else if (n==2) {
          and = AND();
        and.a <== in[0];
        and.b <== in[1];
        out <== and.out;
    } else {
        and = AND();
    var n1 = n\2;
        var n2 = n-n\2;
        ands[0] = MultiAND(n1);
        ands[1] = MultiAND(n2);
        for (i=0; i<n1; i++) ands[0].in[i] <== in[i];
        for (i=0; i<n2; i++) ands[1].in[i] <== in[n1+i];
        and.a <== ands[0].out;
        and.b <== ands[1].out;
        out <== and.out;
    }
}
```

The main component

In order to start the execution, an initial component has to be given. By default, the name of this component is “main”, and hence the component main needs to be instantiated with some template.

This is a special initial component needed to create a circuit and it defines the global input and output signals of a circuit. For this reason, compared to the other components, it has a special attribute: the list of public input signals. The syntax of the creation of the main component is:

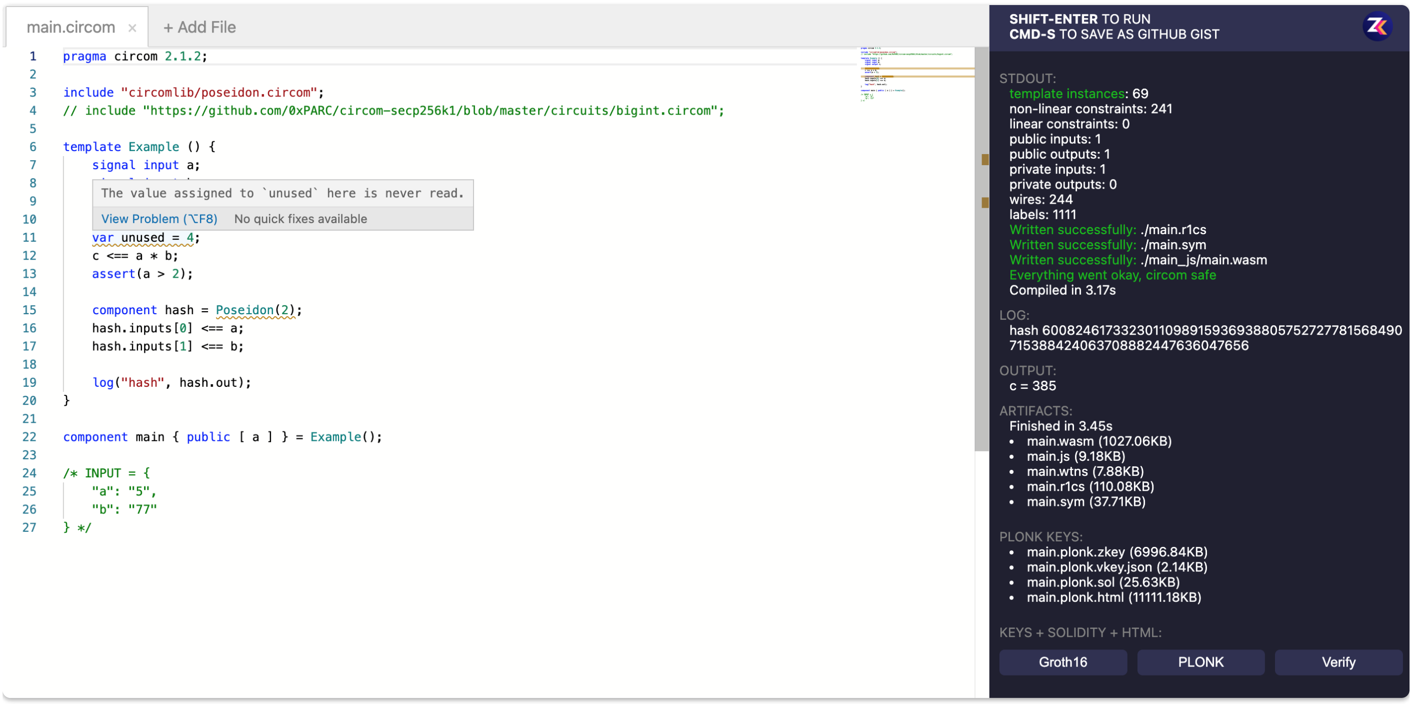
```
  component main {public [signal_list]} = tempid(v1,...,vn);
```

```
  pragma circom 2.0.0;

template A(){
    signal input in1;
    signal input in2;
    signal output out;
    out <== in1 * in2;
}

component main {public [in1]}= A();
```

Useful Tool

zkRepl. from 0xPARC

REPL for circom



```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";
// include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";

template Example () {
    signal input a;
    The value assigned to `unused` here is never read.

    View Problem (⌥F8)    No quick fixes available
    var unused = 4;
    c <== a * b;
    assert(a > 2);

    component hash = Poseidon(2);
    hash.inputs[0] <== a;
    hash.inputs[1] <== b;

    log("hash", hash.out);
}

component main { public [ a ] } = Example();

/* INPUT = {
    "a": "5",
    "b": "77"
} */
```

```
STDOUT:
  template instances: 69
  non-linear constraints: 241
  linear constraints: 0
  public inputs: 1
  public outputs: 1
  private inputs: 1
  private outputs: 0
  wires: 244
  labels: 1111
  Written successfully: ./main.r1cs
  Written successfully: ./main.sym
  Written successfully: ./main_js/main.wasm
  Everything went okay, circom safe
  Compiled in 3.17s
LOG:
  hash 6008246173323011098915936938805752727781568490
  715388424063708882447636047656
OUTPUT:
  c = 385
ARTIFACTS:
  Finished in 3.45s
  •  main.wasm (1027.06KB)
  •  main.js (9.18KB)
  •  main.wtns (7.88KB)
  •  main.r1cs (110.08KB)
  •  main.sym (37.71KB)

PLONK KEYS:
  •  main.plonk.zkey (6996.84KB)
  •  main.plonk.vkey.json (2.14KB)
  •  main.plonk.sol (25.63KB)
  •  main.plonk.html (11111.18KB)

KEYS + SOLIDITY + HTML:
```

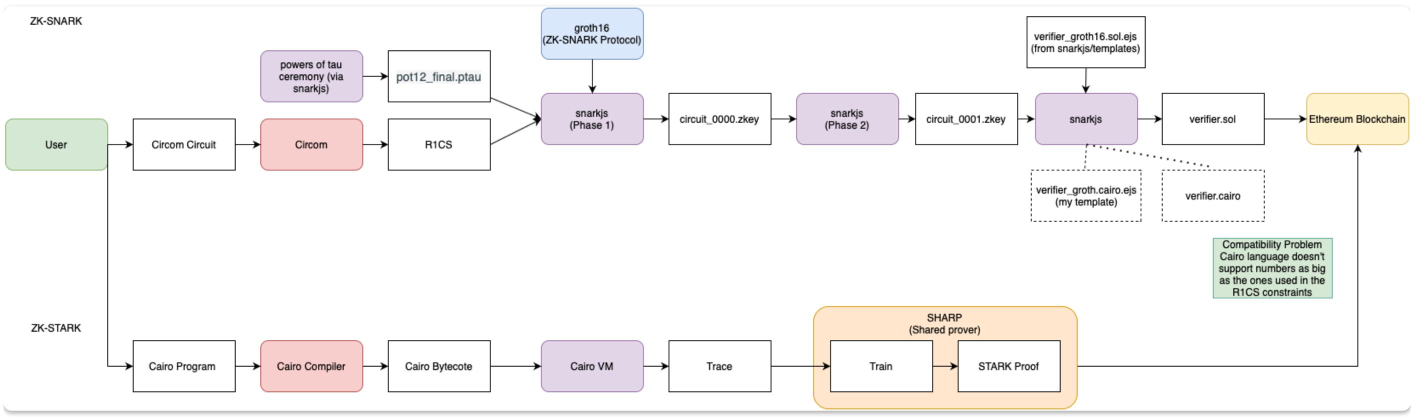
Circom -> Cairo

See repo and take note of the caveats

Allows circom projects to be verified on Ethereum by exporting to Cairo

First write and compile a circuit and compute the witness through circom, then generate a validation key through snarkjs (this process is properly explained at https://docs.circom.io/getting-started/installation/), this will yield a .zkey, which we can use to generate a solidity verifier through the command:

```
  snarkjs zkey export solidityverifier [name of your key].zkey
[nme of the verifier produced]
```



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

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 $B|A$

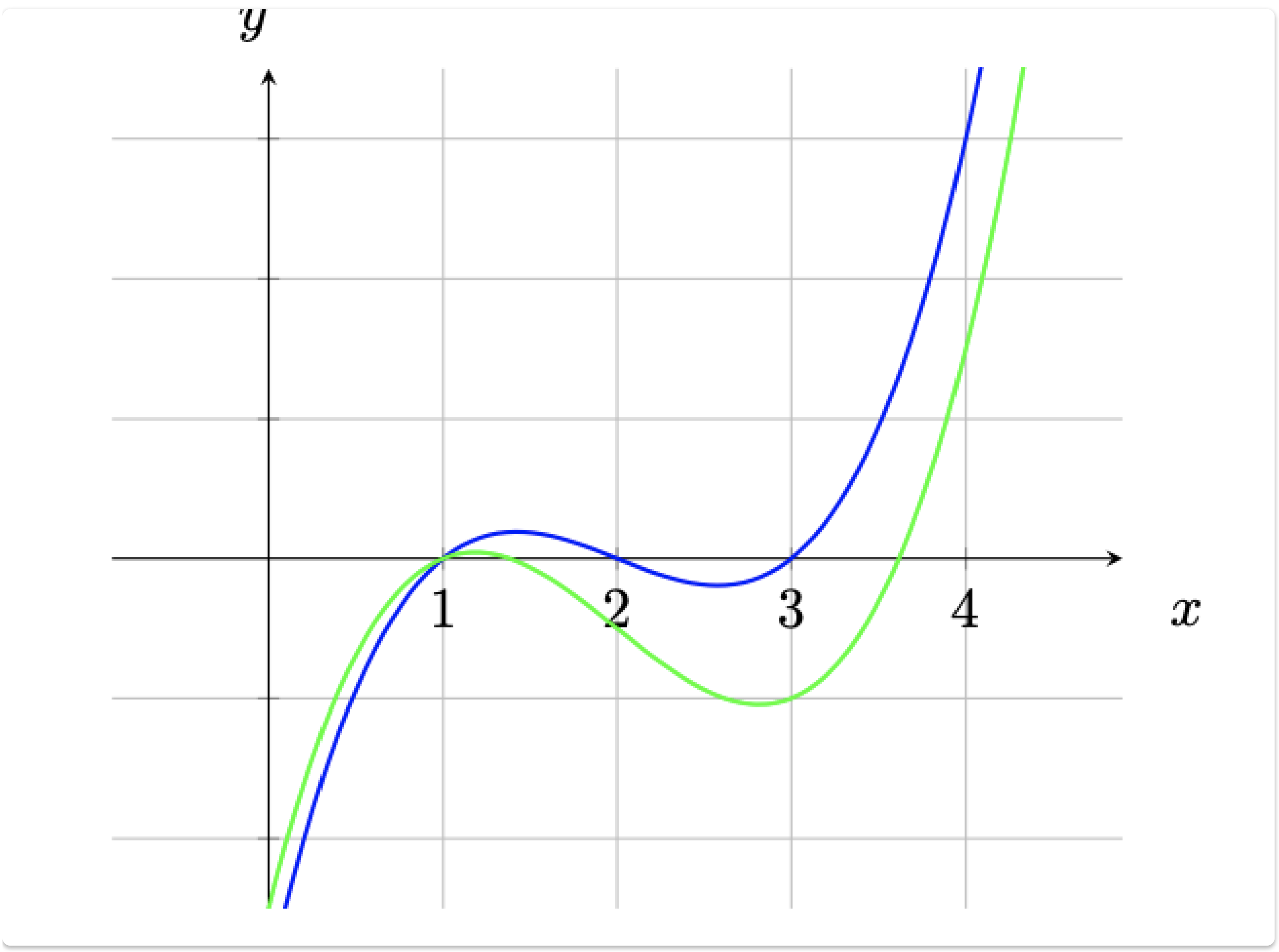
If one root r of a polynomial $P(x)$ of degree n is known then polynomial long division can be used to factor $P(x)$ into the form

$(x-r)(Q(x))$
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
$x^q = x$

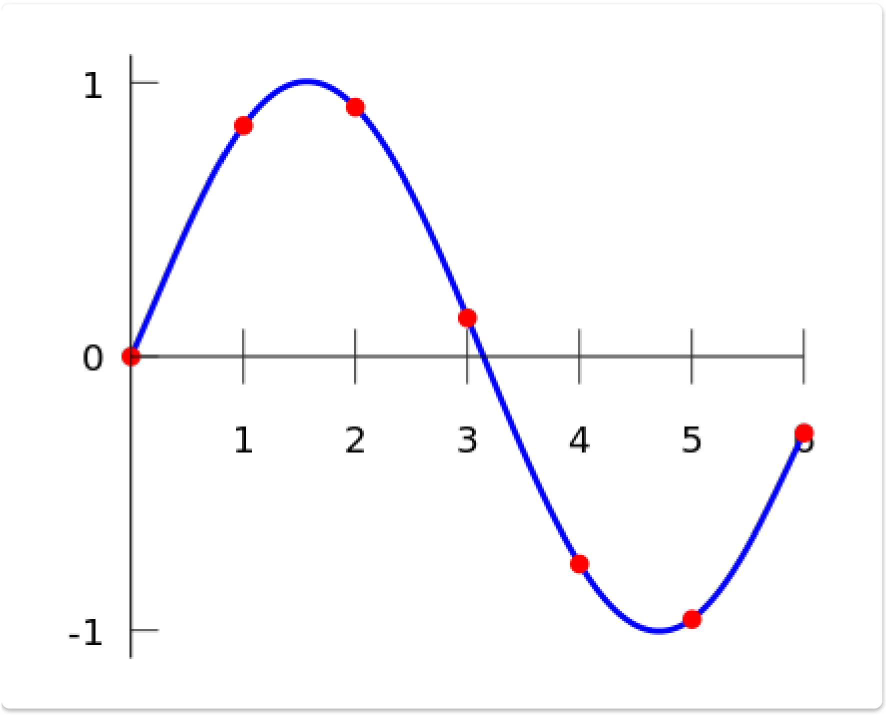
The polynomials X^q and X take the same values at all points, but do not have the same coefficients.

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

2. Point value form
   $(x_1, y_1), (x_2, y_2), \ldots$

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

Polynomials in ZKPs

If a prover claims to know some polynomial (no matter how large its degree is) that the verifier also knows, they can follow a simple protocol to verify the statement:
• Verifier chooses a random value for x and evaluates his polynomial locally
• Verifier gives x to the prover and asks to evaluate the polynomial in question
• Prover evaluates her polynomial at x and gives the result to the verifier
• Verifier checks if the local result is equal to the prover's result, and if so then the statement is proven with a high confidence

In general, there is a rule that if a polynomial P is zero across some set $S = x1, x2 ... xn$ then it can be expressed as
$P(x) = Z(x) * H(x)$, where
$Z(x) = (x - x1) * (x - x2) * ... * (x - xn)$ and
$H(x)$ is also a polynomial.

In other words, any polynomial that equals zero across some set is a (polynomial) multiple of the simplest (lowest-degree) polynomial that equals zero across that same set.

Homomorphic Hiding

(Taken from the ZCash explanation)

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 explanation.

"PolynomialCommitments#Polynomial Commitments" is not created yet. Click to create.

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) $P_i..P_k$: polynomials
4. Verifier provides $z \in 0,..p-1$
5. Prover provides evaluations of polynomials: $P_1(z)...P_k(z)$
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 ~ 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 $0,..p-1$, 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 ?

1. Polynomials are good error correcting codes

If we have polynomials of degree d over an encoding domain D, and two messages $m1$ and $m2$, then $m1$ and $m2$ 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.

2. Have efficient batch zero testing
   This also helps with succinctness

Imagine we want to prove that a large degree polynomial $P(x)$ (degree ~ 10 million) evaluates to zero at points $1..1$ 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 = $1....10^6$

define V as the unique polynomial that vanishes on these points
i.e.
$(x-1)(x-2)(x-3)....$
the degree of V = size of S

this is good because

$P(x)$ vanishes on S iff
there exists $P'(x)$ such that
1. $P(x) = P'(x).V(x)$
2. degree of P' = degree of P - size of S

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

3. 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 as $C(x) = x.(x-1)$

> 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.
> $C(P_1(x)) = P_1(x).(P_1(x) - 1)$
>
> 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 before we can say that there is some polynomial $P'(x)$ such that
>
> $C(P_1(x)) = P'(x).V(x)$
>
> 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.

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 if efficiently.

From our QAP we have

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

and we define the polynomial P
$P := L \cdot R - O$
Defining the target polynomial $V(x) := (x-1) \cdot (x-2) \ldots$,
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
$P = P'.V$

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 (c1,…,cm) (c1,…,cm).
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.

Homomorphic Hiding Review

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 $E(1), E(z_1), \ldots, E(z_d)$
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 $1, z_1, \ldots, z_d$)

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).

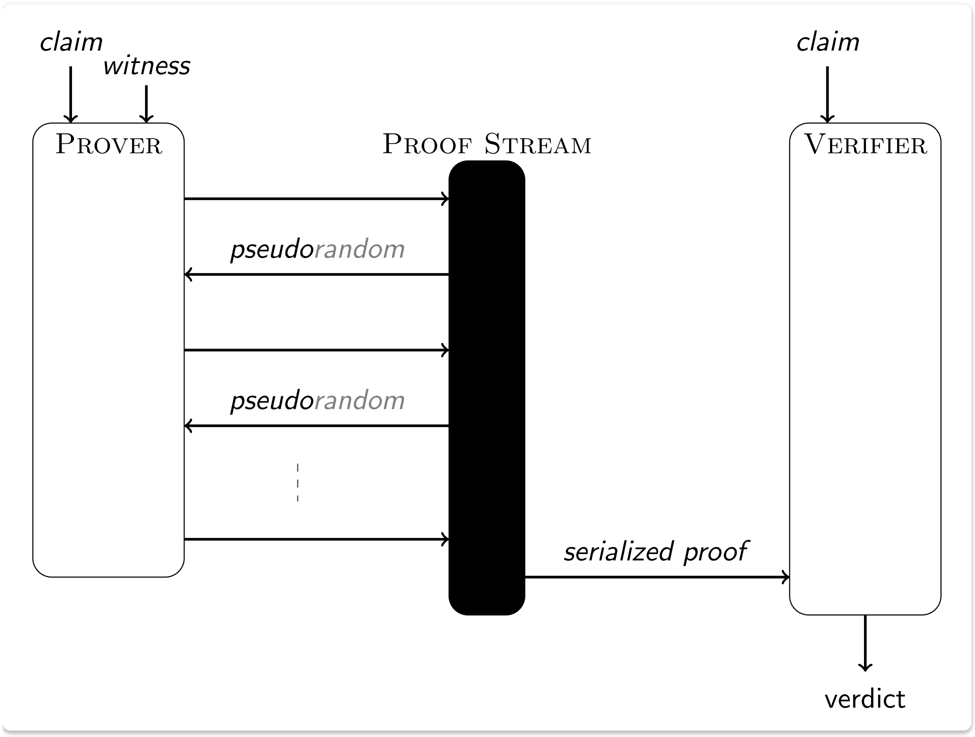
So, Victor must only issue the challenge after Peggy has already fixed her randomness. This is why Peggy first *commits* to her randomness, and implicitly reveals it only after the challenge, when she uses that value to compute the proof. That ensures two things:

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

Fiat-Shamir heuristic

See https://aszepieniec.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