Lesson 15

Week 4

Lesson 13 - What are ZK EVMs part 3 - proving system: zk algorithm/ASIC optimisation
Lesson 14 - zkEVM security
Lesson 15 - Overview of Proving Systems
Lesson 16 - SNARK implementation

Today's topics

- General ZKP Theory
- SNARK, PLONK, STARK Overview
- Bulletproofs Overview
- STARK Overview
- Plonkish protocols
- Folding Schemes

Proofs

What do we require of a proof ?

- Completeness: there exists an honest prover P that can convince the honest verifier V of any correct statement with very high probability.
- Soundness: even a dishonest prover P running in super-polynomial time cannot convince an honest verifier V of an incorrect statement. Note: P does not necessarily have to run in polynomial time, but V does.

To make our proof zero knowledge we also need 'zero knowledginess'

Much effort is put into proving systems to ensure that they are sound.

In 1992 we had the invention of Probabilistic Checkable Proofs which almost 'solved' the area, except that there was a bottleneck with prover time, since then we have seen improvements with linear PCPs and IOPs.

Zero Knowledge Proof Timeline

Changes have occurred because of

- Improvements to the cryptographic primitives (improved curves or hash functions for example)

- A fundamental change to the approach to zero knowledge
  See the excellent blog post from Starkware : [The Cambrian Explosion]

1984 : Goldwasser, Micali and Rackoff - Probabilistic Encryption.
1989 : Goldwasser, Micali and Rackoff - The Knowledge Complexity of Interactive Proof Systems
1991 O Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. Preliminary version in 1986. (Graph colouring problem)

....

2006 Groth, Ostrovsky and Sahai introduced pairing-based NIZK proofs, yielding the first linear size proofs based on standard assumptions.

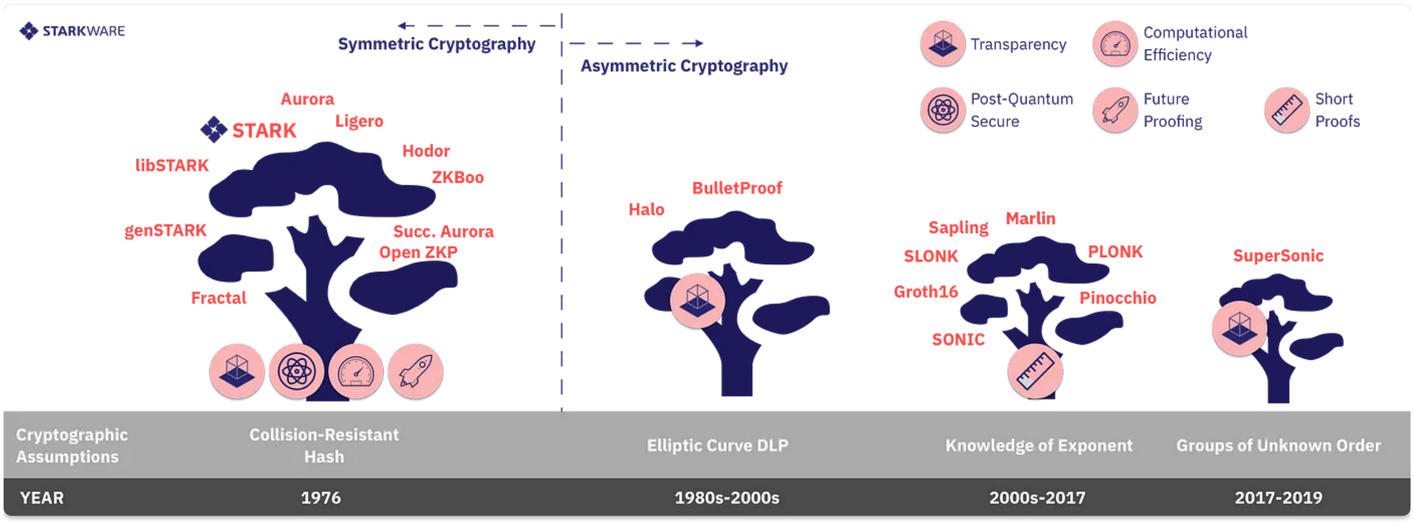
2010 Groth combined these techniques with ideas from interactive zero-knowledge arguments to give the first constant size NIZK arguments.

2016 : Jens Groth - On the Size of Pairing-based Non-interactive Arguments

From [Matthew Green]

Prior to Goldwasser et al., most work in this area focused the soundness of the proof system. That is, it considered the case where a malicious Prover attempts to ‘trick’ a Verifier into believing a false statement. What Goldwasser, Micali and Rackoff did was to turn this problem on its head. Instead of worrying only about the Prover, they asked: what happens if you don’t trust the Verifier?

ZKP Ecosystem



from

The Cambrian Explosion

Proving Systems

A statement is a proposition we want to prove. It depends on:

- Instance variables, which are public.
- Witness variables, which are private.

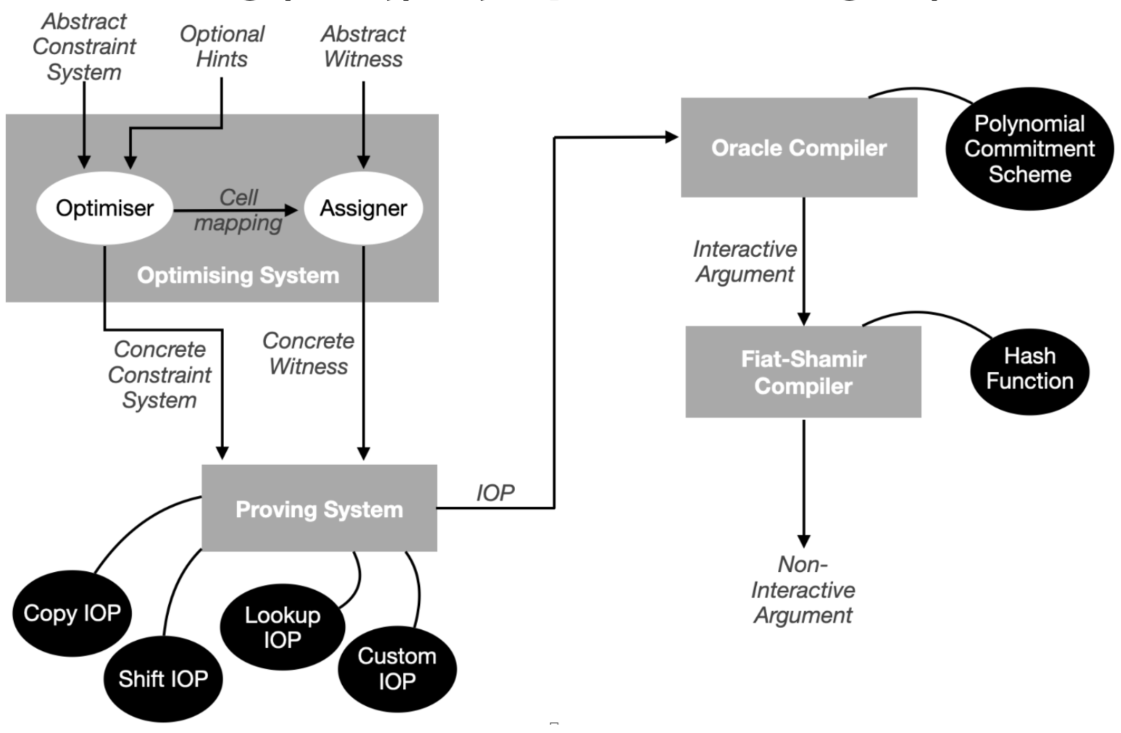
Given the instance variables, we can find a short proof that we know witness variables that make the statement true (possibly without revealing any other information).

> **SNARK overview**
>
> The important letter from the acronym is S for Succinct
> A SNARK is a triple (S, P, V)
> $S(C)$ provides public parameters pp and vp (a setup phase)
> $P(pp, x, w)$ is a short proof π
> $V(vp, x, \pi)$ - succinct verification
>
> C here is a circuit, a representation of the program or statement we wish to prove.
>
> We require
> len(π) = O_λ (log (|C|))
> time(V) = O_λ (|x|, log (|C|))
>
> Since verification needs to be succinct, vp provides a summary of the circuit.

A typical SNARK arithmetisation is shown in the PLONK process.

> Plonk Process overview

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



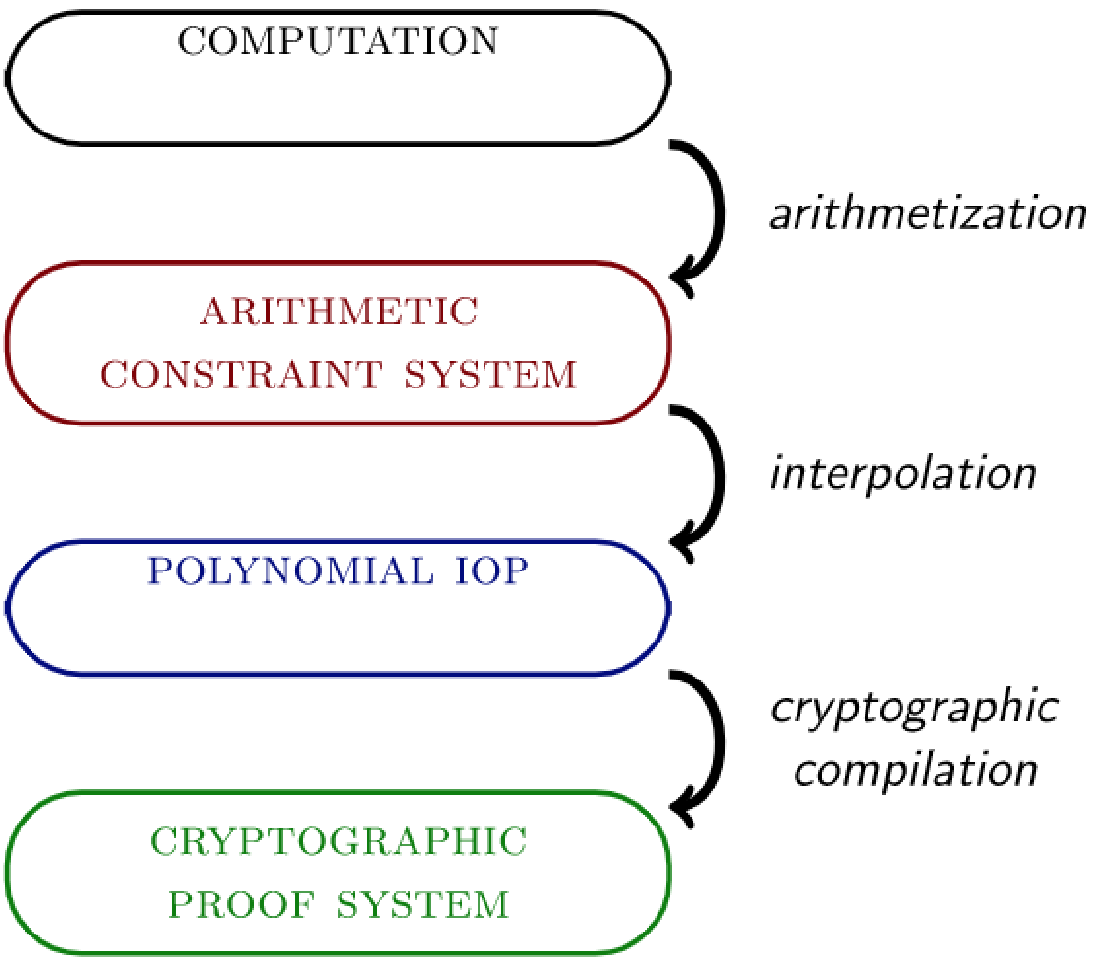
We shall go through the details of SNARKS in a later lesson.

Stark Introduction

STARKS are
scalable transparent arguments of knowledge

They have much simpler cryptographic assumptions than SNARKS

Overview of the Stark process


COMPUTATION
arithmetization
ARITHMETIC CONSTRAINT SYSTEM
interpolation
POLYNOMIAL IOP
cryptographic compilation
CRYPTOGRAPHIC PROOF SYSTEM

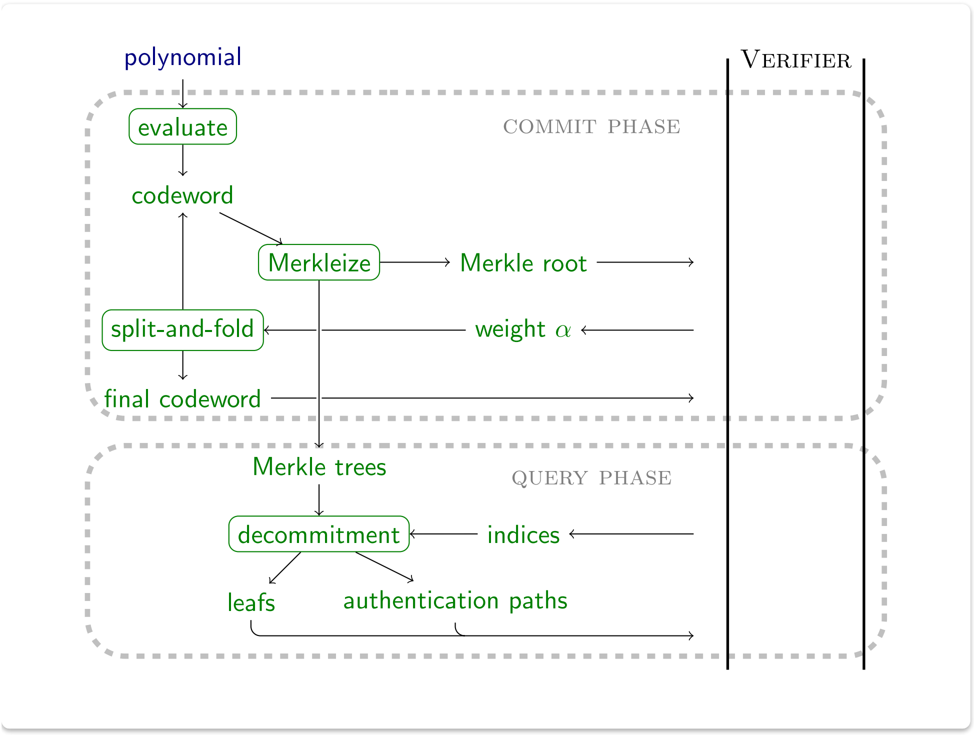
This process is carried out in two steps:

> 1. Creation of an execution trace and polynomial constraints
> 2. Conversion of these two components into a single low-degree polynomial.

FRI

FRI stands for *Fast Reed-Solomon IOP of Proximity*, it is a protocol that establishes that a committed polynomial has a bounded degree.

FRI is complex and much of the processing that makes it up is designed to make the testing feasible and succinct. There is also much processing involved with guarding against various types of attacks that could be made by the prover, and ensuring that everything is carried out in zero knowledge.
It aims to find if a set of points are mostly on a polynomial of low degree and can achieve linear proof complexity and logarithmic verification complexity.



Overview

Bulletproofs are short non-interactive zero-knowledge proofs that require no trusted setup.

See paper : Bulletproofs

- Bullet proofs have short proofs without a trusted setup.
- Their underlying cryptographic assumption is the discrete log problem.
- They are made non interactive using the Fiat-Shamir heuristic.
- One of the paper's authors referred to them as “Short like a bullet with bulletproof security assumptions”
- They were designed to provide confidential transactions for cryptocurrencies.
- They support proof aggregation, so that proving that m transaction values are valid, adds only $O(log(m))$ additional elements to the size of a single proof.
- Pederson commitments are used for the inputs
- They do not require pairings and work with any elliptic curve with a reasonably large subgroup size
- The verifier cost scales linearly with the computation size.

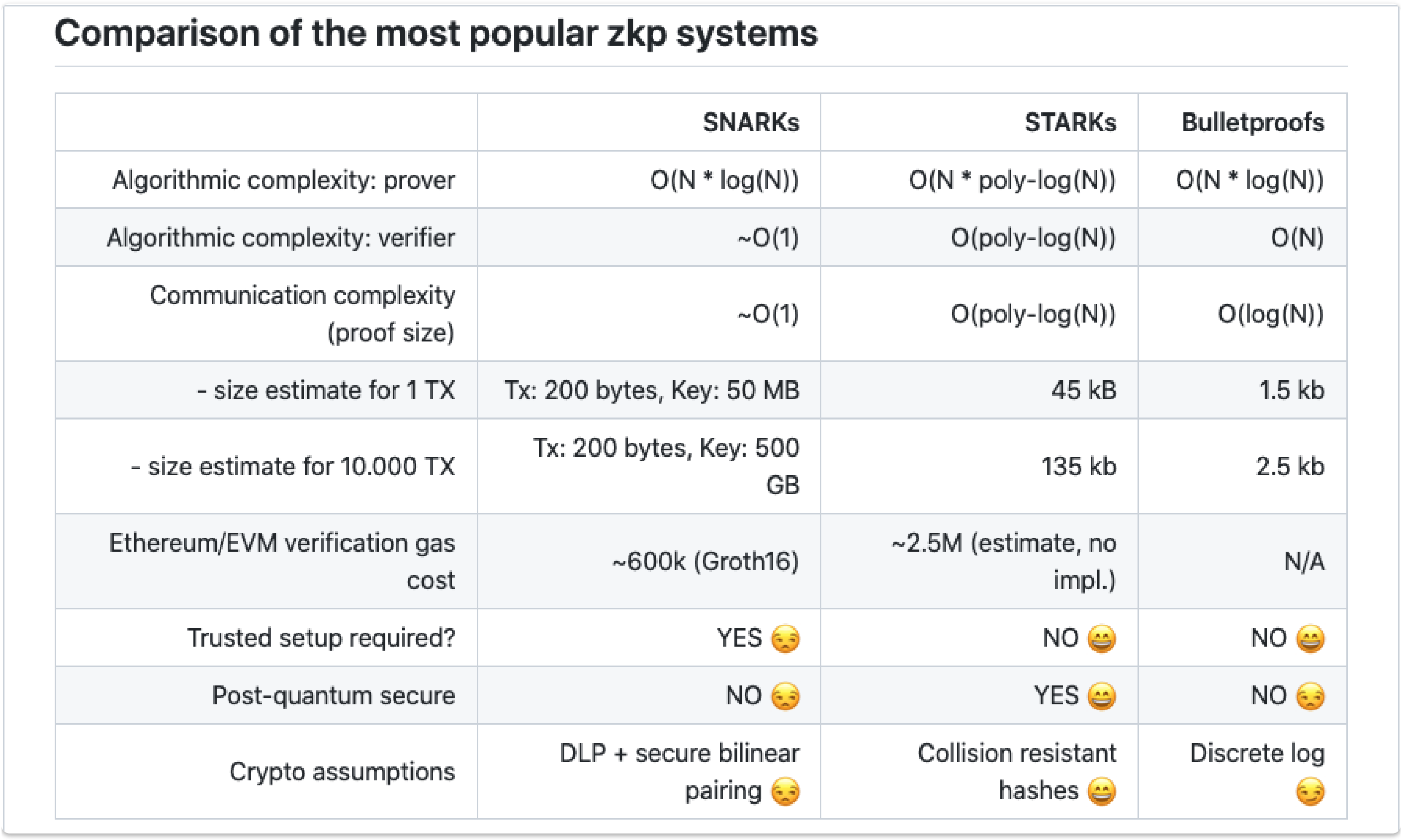
Bulletproofs were based on ideas from Groth16 SNARKS, but changed various aspects.

- They give a more compact version of the inner product argument of knowledge
- Allow construction of a compact rangeproof using such an argument of knowledge
- They generalise this idea to general arithmetic circuits.

Some use cases for bulletproofs

- Range proofs
- Merkle proofs (accumulators)
- Proof of Solvency

Comparison



Comparison of the most popular zkp systems

	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	DLP + secure bilinear pairing 😒	Collision resistant hashes 😄	Discrete log 😒

Framework	Arithmetization	Commitment Scheme	Field	Other Configs
Circom + snarkjs / rapidsnark	R1CS	Groth16	BN254 scalar	
gnark	R1CS	Groth16	BN254 scalar	
Arkworks	R1CS	Groth16	BN254 scalar	
Halo2 (KZG)	Plonkish	KZG	BN254 scalar	
Plonky2	Plonk	FRI	Goldilocks	blowup factor = 8 proof of work bits = 16 query rounds = 28 num_of_wires = 60 num_routed_wires = 60
Starky	AIR	FRI	Goldilocks	blowup factor = 2 proof of work bits = 10 query rounds = 90
Boojum	Plonk	FRI	Goldilocks	

Plonkish protocols

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

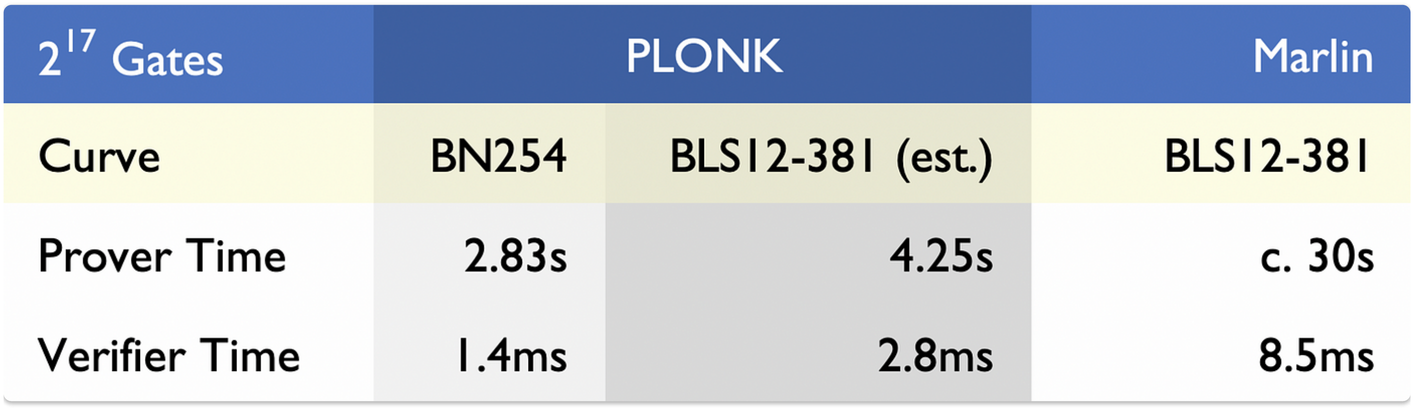
Before PLONK

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 $G, G \cdot s, G \cdot s^2 \ldots G \cdot s^n$, as well as $G2 \cdot s$, where G and $G2$ 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.

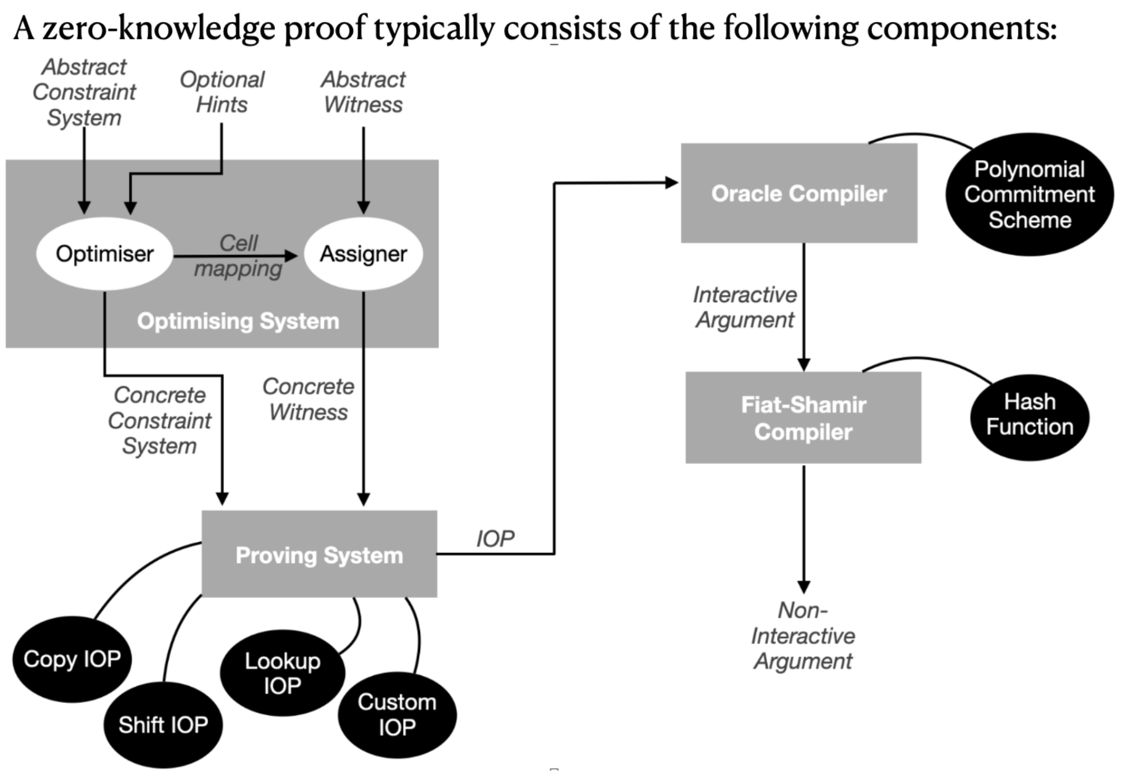
In January 2019 Mary Maller, Sean Bowe et al released SONIC that has a universal setup, with just one setup, it could validate any conceivable circuit (up to a predefined level of complexity).
This was unfortunately not very efficient, PLONK managed to optimise the process to make the proof process feasible.



2^{17} 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

See PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge
Also see Understanding PLONK

Plonk Process overview

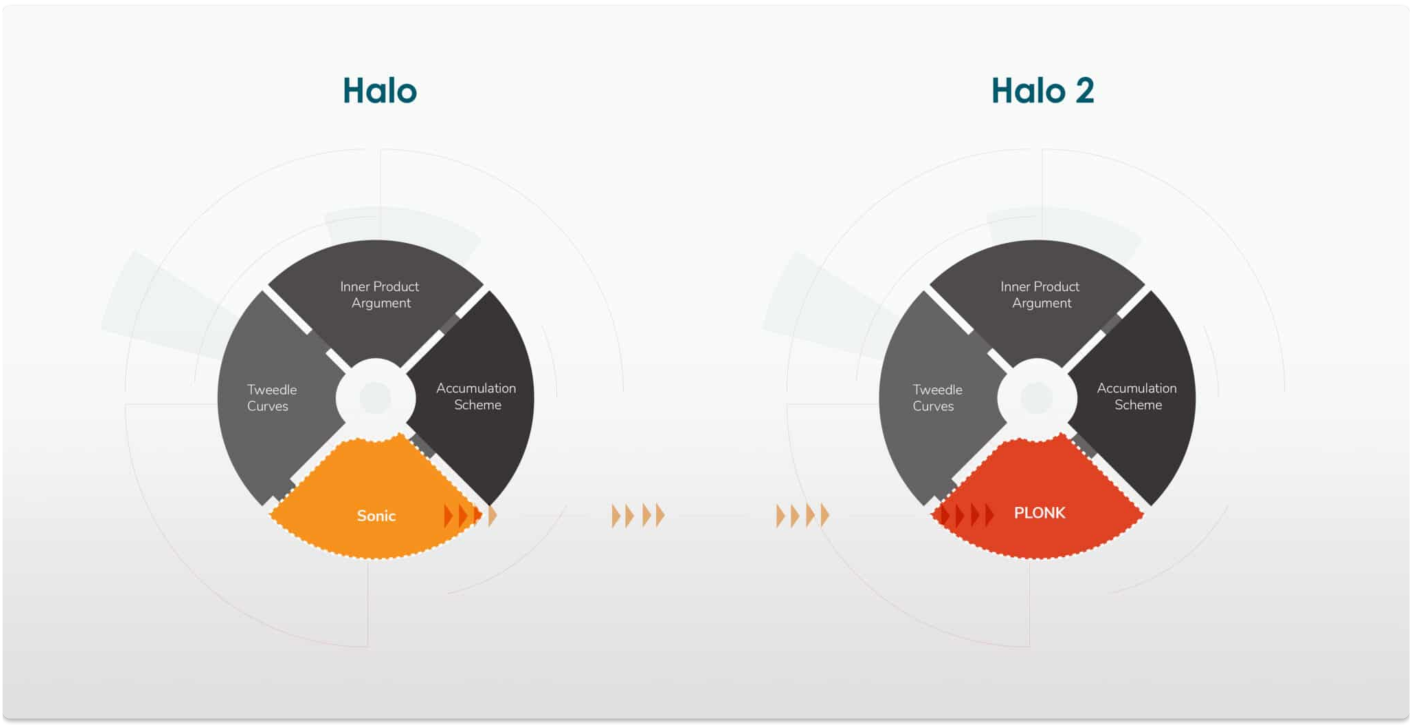

A zero-knowledge proof typically consists of the following components:

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:

We will cover PLONK in more detail in a later lesson.

> **Halo2**
>
> See tutorial
>
> See documentation
>
> See Halo2 Book
>
> Halo 2 is a proving system that combines the Halo recursion technique with an arithmetisation based on PLONK, and a polynomial commitment scheme based around the Inner Product Argument
>
> **History**
>
> 
>
>
> Chips
>
> Using our API, we define chips that "know" how to use particular sets of custom gates. This creates an abstraction layer that isolates the implementation of a

high-level circuit from the complexity of using custom gates directly.

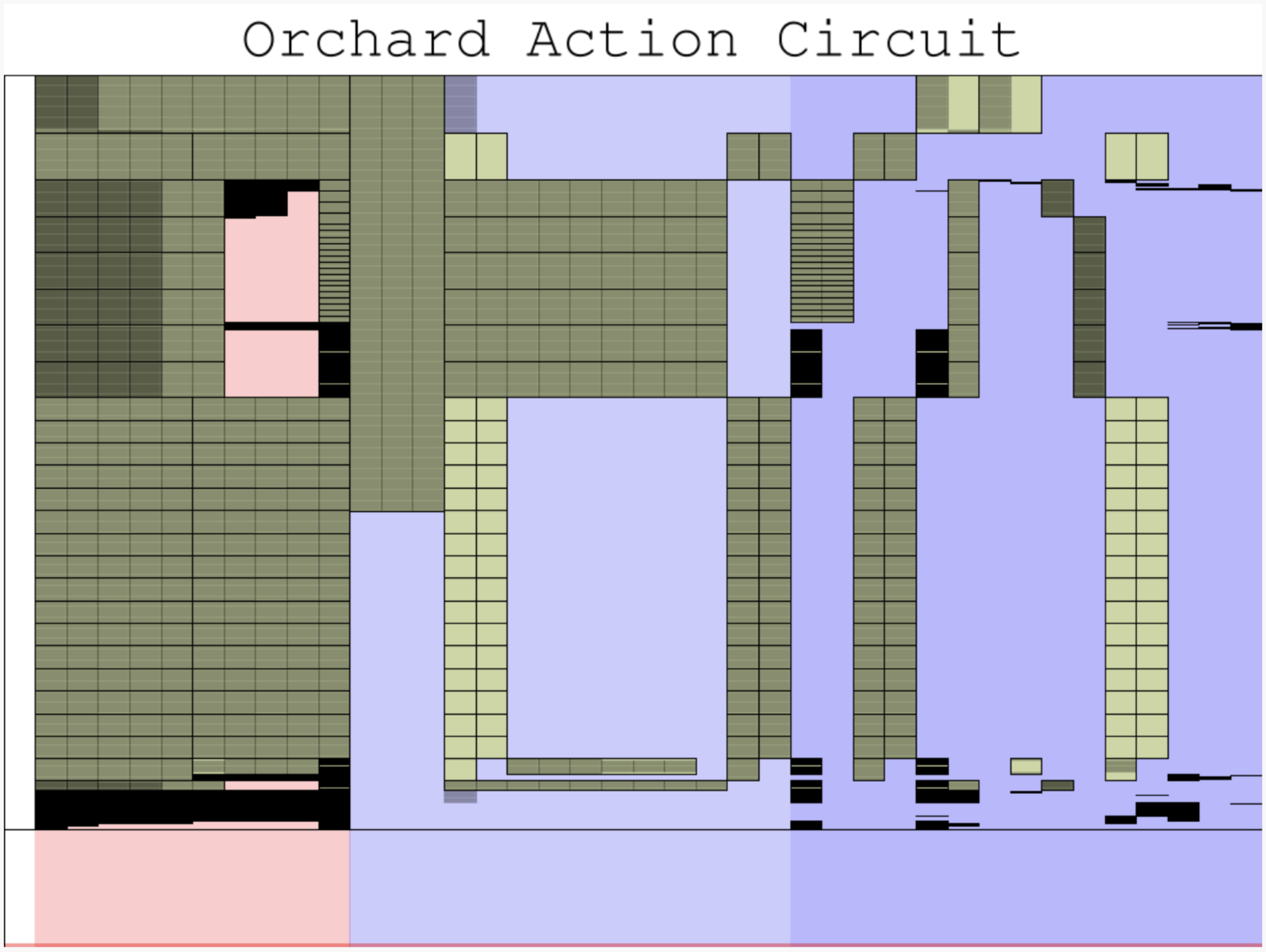
Example Simple Circuit

language-rust

```
trait NumericInstructions<F: Field>: Chip<F>
{
    /// Variable representing a number.
    type Num;

    /// Loads a number into the circuit as a private input.
    fn load_private(&self, layouter: impl Layouter<F>, a: Value<F>) -> Result<Self::Num, Error>;

    /// Loads a number into the circuit as a fixed constant.
    fn load_constant(&self, layouter: impl Layouter<F>, constant: F) -> Result<Self::Num, Error>;

    /// Returns `c = a * b`.
    fn mul(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    /// Exposes a number as a public input
to the circuit.
    fn expose_public(
        &self,
        layouter: impl Layouter<F>,
        num: Self::Num,
        row: usize,
    ) -> Result<(), Error>;
}
```

Halo 2 circuits are two-dimensional: they use a grid of "cells" identified by columns and rows, into which values are assigned.
Constraints on those cells are grouped into "gates", which apply to every row simultaneously, and can refer to cells at relative rows.
To enable both low-level relative cell references in gates, and high-level layout optimisations, circuit developers can define "regions" in which assigned cells will preserve their relative offsets.

Example from ZCash



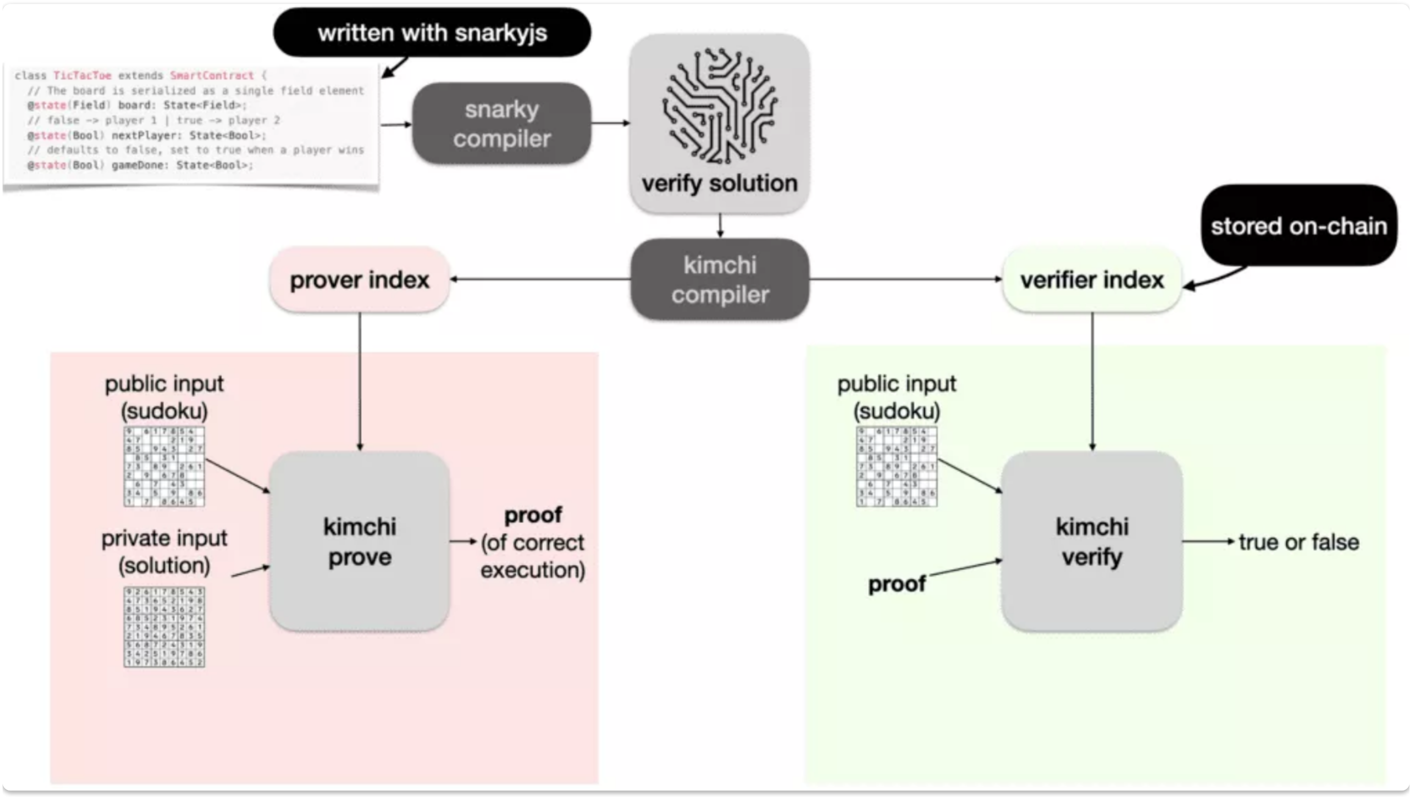
In the example circuit layout pictured, the columns are indicated with different backgrounds.
The instance column in white;
advice columns in red;
fixed columns in light blue; and
selector columns in dark blue.
Regions are shown in light green, and assigned cells in dark green or black.

Column Types

- Instance columns contain per-proof public values, that the prover gives to the verifier.

> - Advice columns are where the prover assigns private (witness) values, that the verifier learns zero knowledge about.
> - Fixed columns contain constants used by every proof that are baked into the circuit.
> - Selector columns are special cases of fixed columns that can be used to selectively enable gates.

Pickles and Kimchi



```
class TicTacToe extends SmartContract {
  // The board is serialized as a single field element
  @state(Field) board: State<Field>;
  // false -> player 1 | true -> player 2
  @state(Bool) nextPlayer: State<Bool>;
  // defaults to false, set to true when a player wins
  @state(Bool) gameDone: State<Bool>;
```

Pickles

Pickles has two components:

- core zk-SNARK,
- developer toolkit (containing a wide array of library functionality and the Pickles Inductive Proof System)

Kimchi

The heart of the Mina proving system is Kimchi, this is the protocol that creates the proofs.
Kimchi is based on the Plonk family of zkSNARK proof systems.

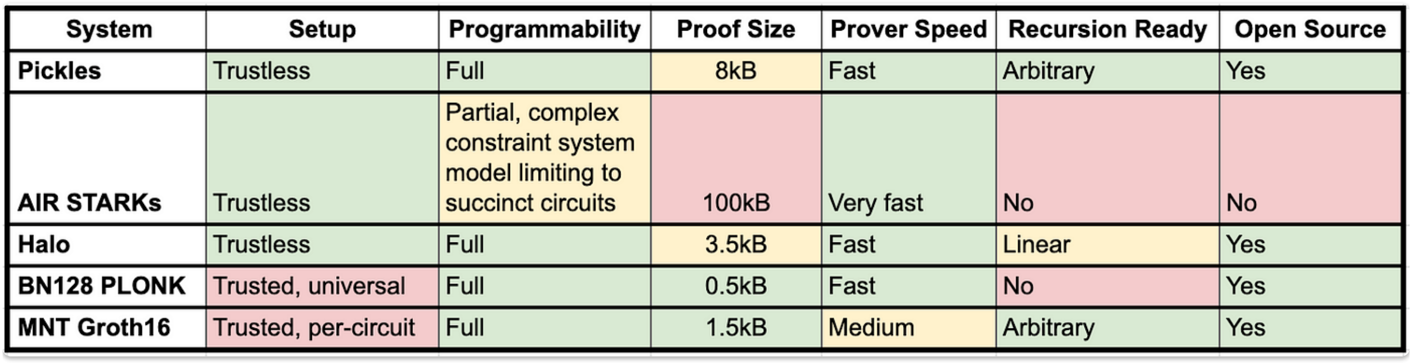
One notable improvement over Plonk is that Kimchi doesn't require a trusted setup.

This is achieved by adding a polynomial commitment to the setup (similar to Bulletproofs)

Other Improvements in Kimchi

- The gates used in the circuit have changed from being generic gates (which could represent any functionality) to specific gates to target known functionality such as Posiedon Hashes.
- More than 2 inputs are allowed for gates.
- The use of lookup tables for public data or simple boolean operations.

How does Pickles compare to other proof systems ?



System	Setup	Programmability	Proof Size	Prover Speed	Recursion Ready	Open Source
Pickles	Trustless	Full	8kB	Fast	Arbitrary	Yes
AIR STARKs	Trustless	Partial, complex constraint system model limiting to succinct circuits	100kB	Very fast	No	No
Halo	Trustless	Full	3.5kB	Fast	Linear	Yes
BN128 PLONK	Trusted, universal	Full	0.5kB	Fast	No	Yes
MNT Groth16	Trusted, per-circuit	Full	1.5kB	Medium	Arbitrary	Yes

Curves used

Mina uses the curves Pallas and Vesta (collectively Pasta).

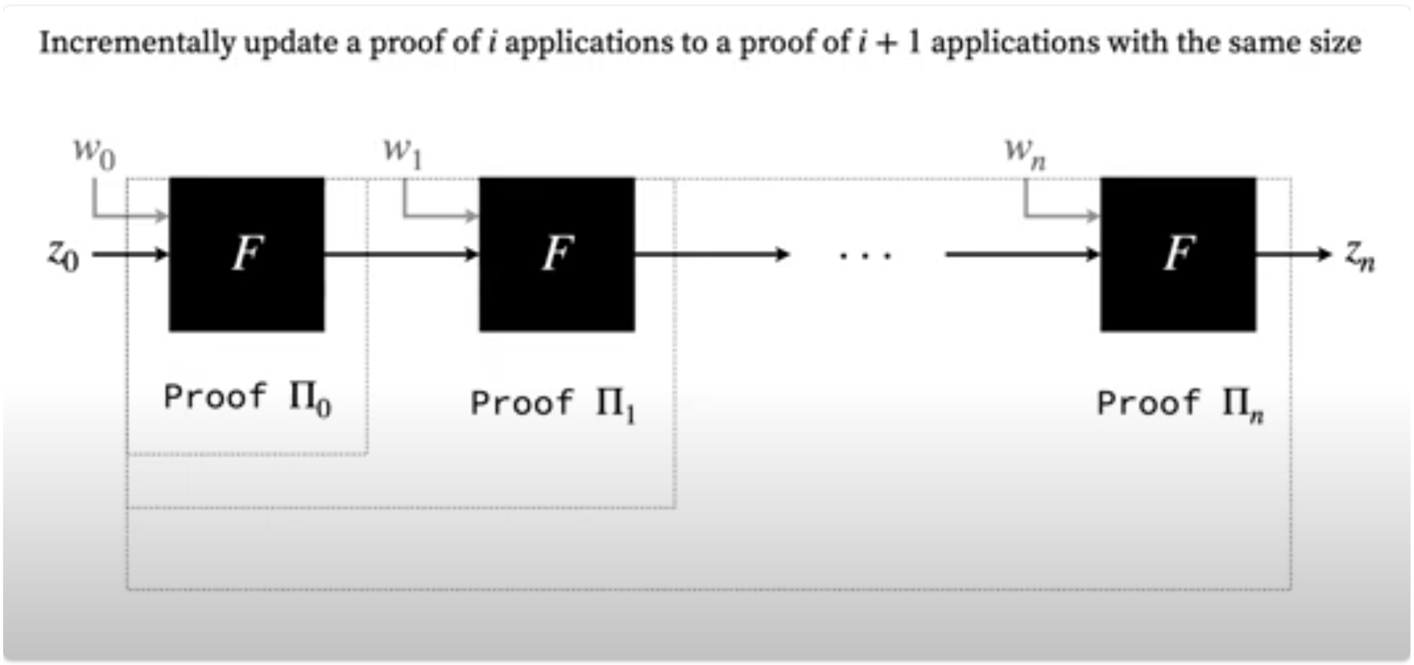
> **Customisable Constraint Systems**
>
> See Paper
> "Customizable constraint system (CCS) is a generalization of R1CS that can simultaneously capture R1CS, Plonkish, and AIR without overheads.
> Unlike existing descriptions of Plonkish and AIR, CCS is not tied to any particular proof system."
> CCS witnesses tend to be smaller than Plonkish ones.

Folding Schemes introduction

With earlier SNARKS if we wanted to use recursion, we needed to have a verifier as part of our circuit, a complex and potentially non optimal technique.

Halo improved on this by allowing some of the verification algorithm to be taken out of the circuit.

Folding schemes take this much further, with virtually all of the verification outside of the circuit.
Folding schemes are often used for IVC



Incrementally update a proof of i applications to a proof of $i + 1$ applications with the same size
w_0
w_1
w_n
z_0
F
F
. . .
F
z_n
Proof Π_0
Proof Π_1
Proof Π_n

Main projects

NOVA

See Paper
See article for an in depth description.
See Video from Justin Drake

Sangria

See paper
A folding scheme for PLONK

Protostar

See paper
This relies on accumulation , a simple yet powerful primitive that enables incrementally verifiable computation (IVC) without the need for recursive SNARKs

The verifier is expressed as a series of equations (more formally, an *algebraic* check). These folding schemes are based on the techniques of Nova and Sangria and introduce optimisations to avoid expensive commitments to cross-terms, especially when dealing with high degree gates.

Supernova

This improves on Nova in the case where we are encoding steps of a VM, originally the prover would end up paying for operations that were supported , even if they were not being invoked.
In Supernova we can reduce this to only paying for the operations that are invoked.

Hypernova

This is Nova using ccs and also using a VDF

Resources

> See awesome repo

> **Applications**
>
> - VDFs (Implementation repo)
> - zk Virtual Machines
> - zk Rollups (transition function is a step)
> - Nova sha256