Extra notes

zkML Bootcamp

Learn about zkML in our dedicated bootcamp.

- Mondays-Thursdays for three weeks
- Learn with an expert in the field
- **Starting on 25th March**

Read the official announcement.

Sign up here

ZK-ML

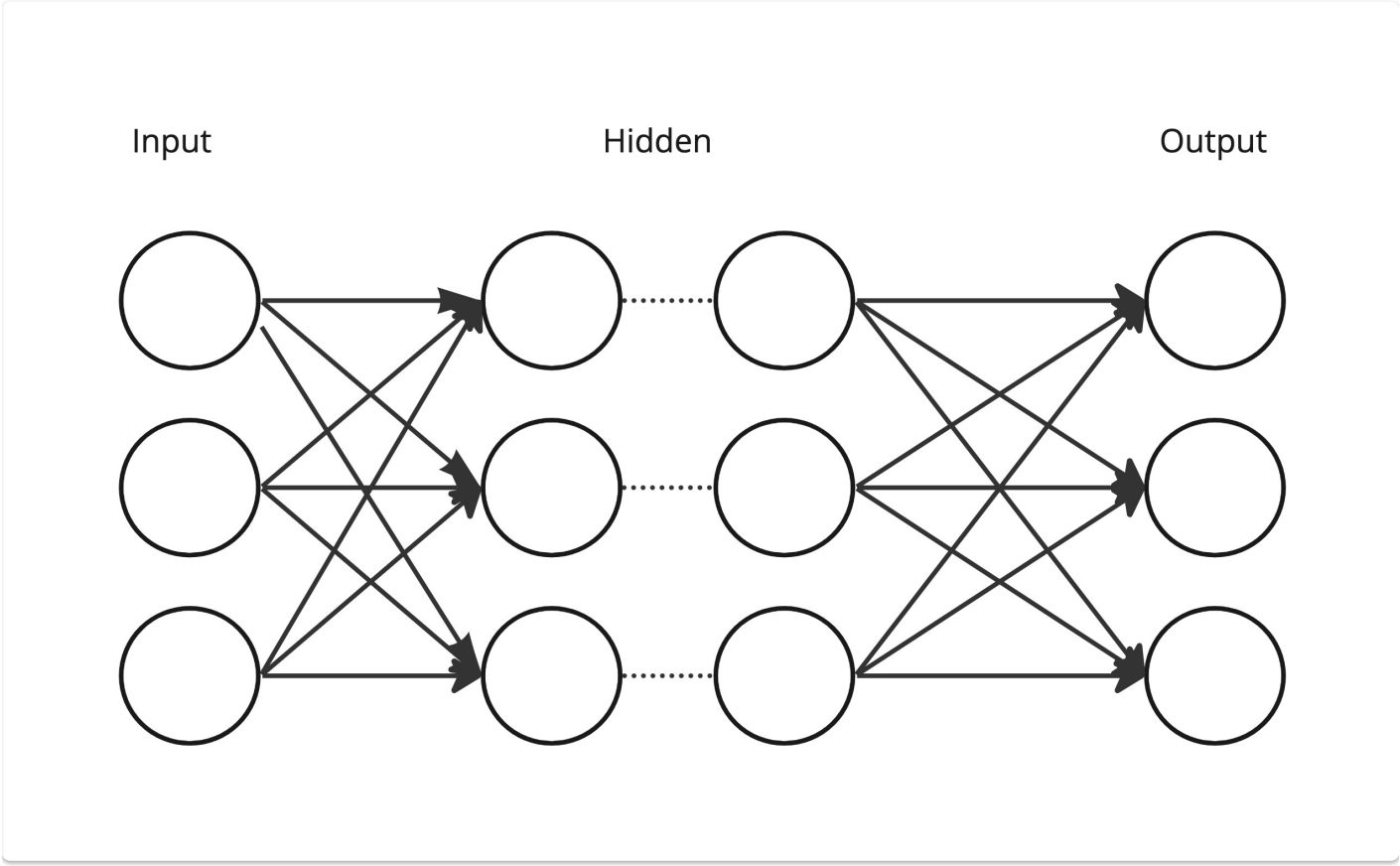
Machine Learning background

Machine learning, a branch of artificial intelligence, focuses on creating and utilizing algorithms that allow computers to independently learn and adapt from data. Through iterative processes, this leads to the optimization of performance. Large-scale language models such as GPT-4 and Bard epitomize the latest advances in natural language processing, using extensive training data to craft text that closely resembles human language.

In a neural network, each node functions as a linear regression model, accepting an input and possessing its own error term, or bias term, along with weights.
These weights are essential as they define the significance that a specific node assigns to a particular segment of the input.
The node's output is then determined, and an activation function applied to this output ascertains if the neural network will activate or

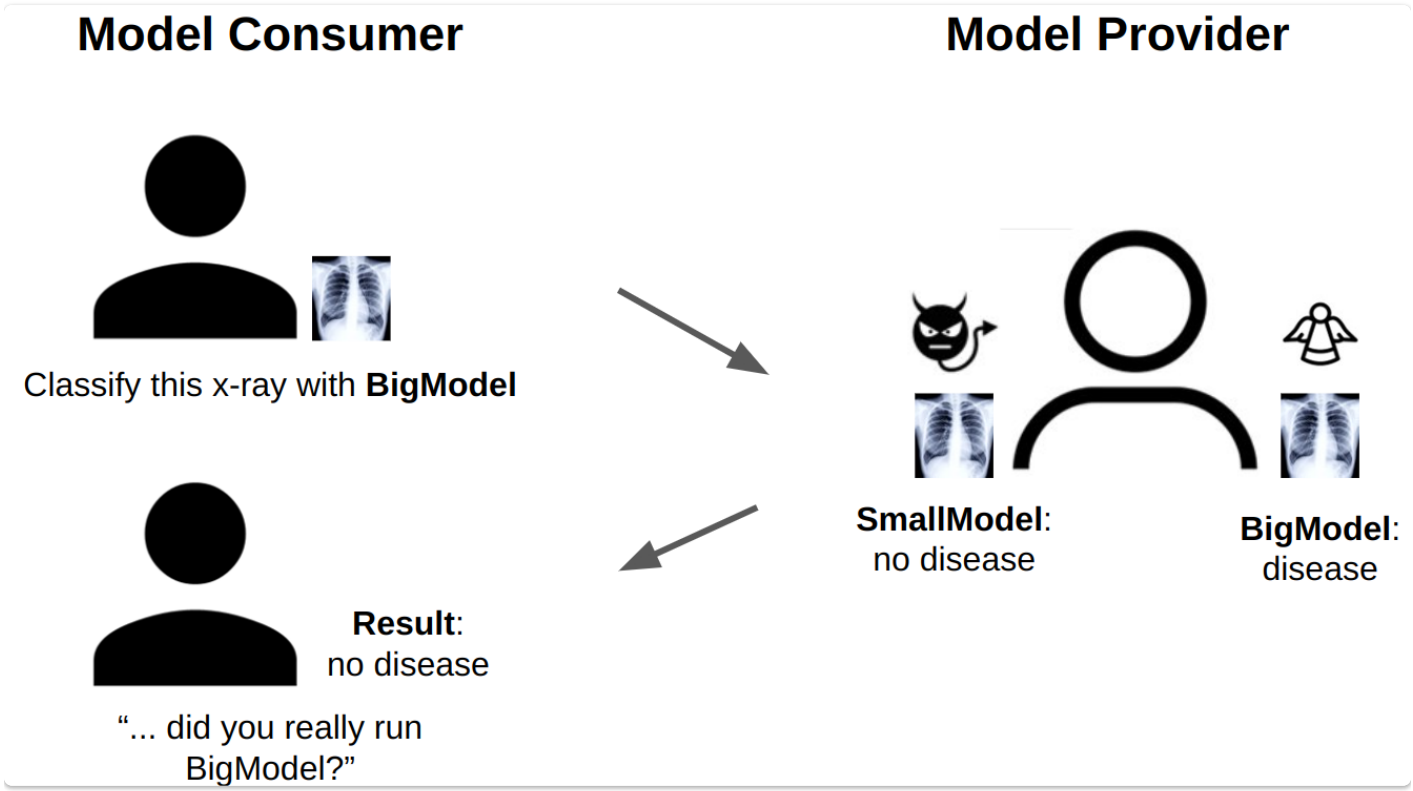
"fire."
Activation occurs if the output value surpasses a set threshold.

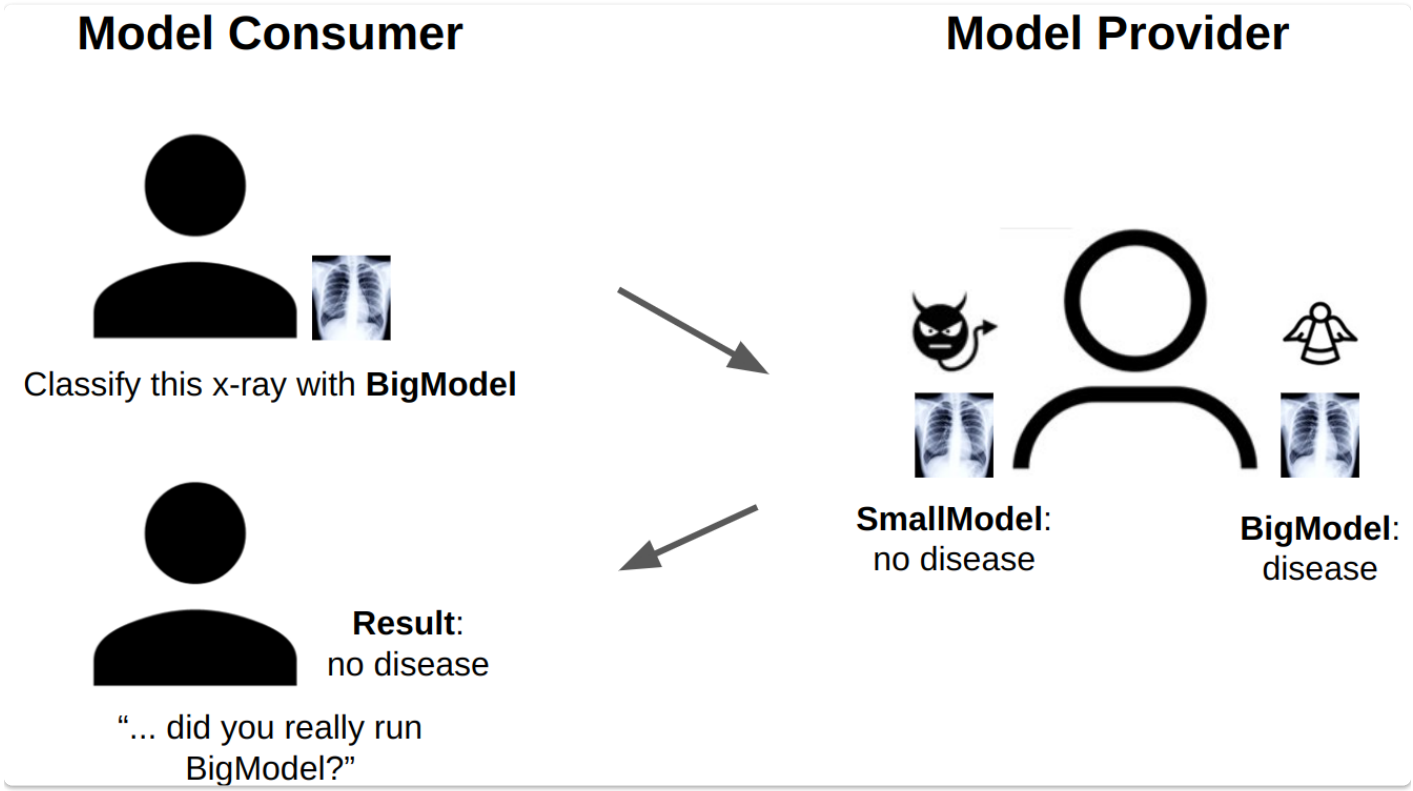
When a node activates or "fires," its output is then used as the input for another node in the network. This process continues with nodes from one layer transmitting data to the succeeding layer, culminating in a chain of data flow. This sequential data transfer from one layer to the next is what gives rise to the concept known as a "feedforward" network.



Training of the network involves 'back propagation' as is many times more expensive than 'feed forward' or inference.

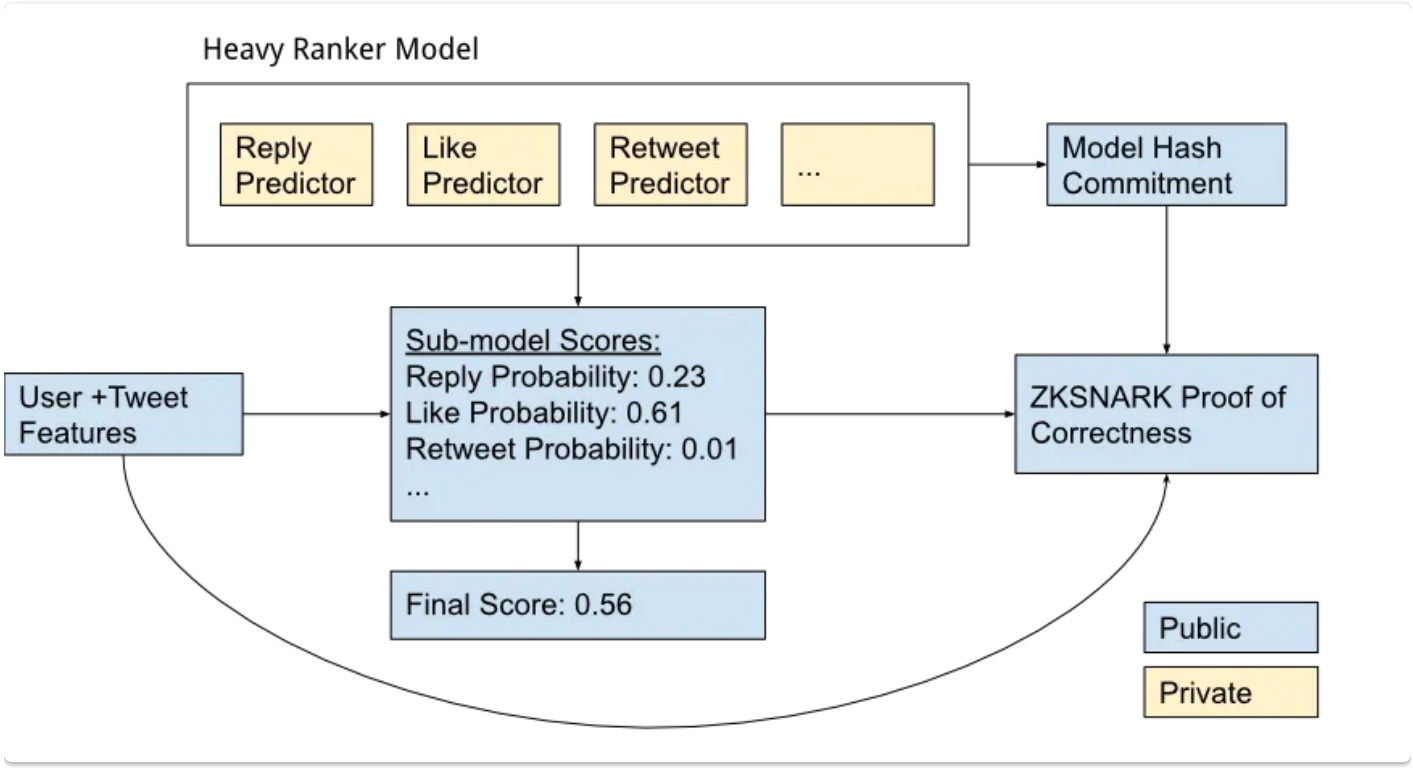
Example use case





Example verifying twitter feeds
See blog

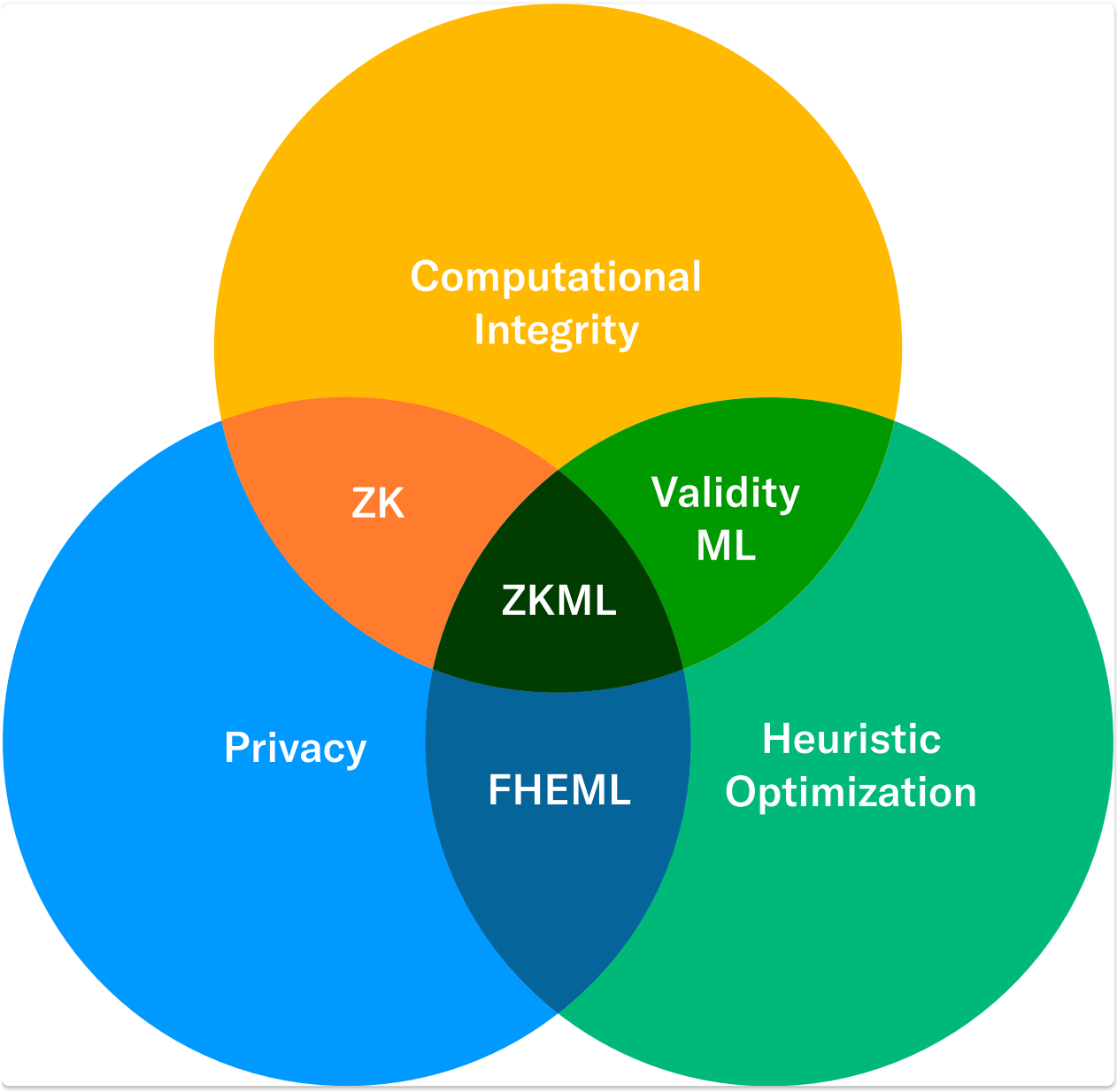
Twitter uses a ML model to select the tweets shown to you, this project uses zkML to confirm that this is running correctly.



Unfortunately using zkML for this is impractical
Proving the Twitter model currently takes 6 *hours* to prove for a *single example* using ezkl.
Verifying the tweets published in one second (~6,000) would cost *~$88,704* on cloud compute hardware.

zkML Introduction

From introduction blog

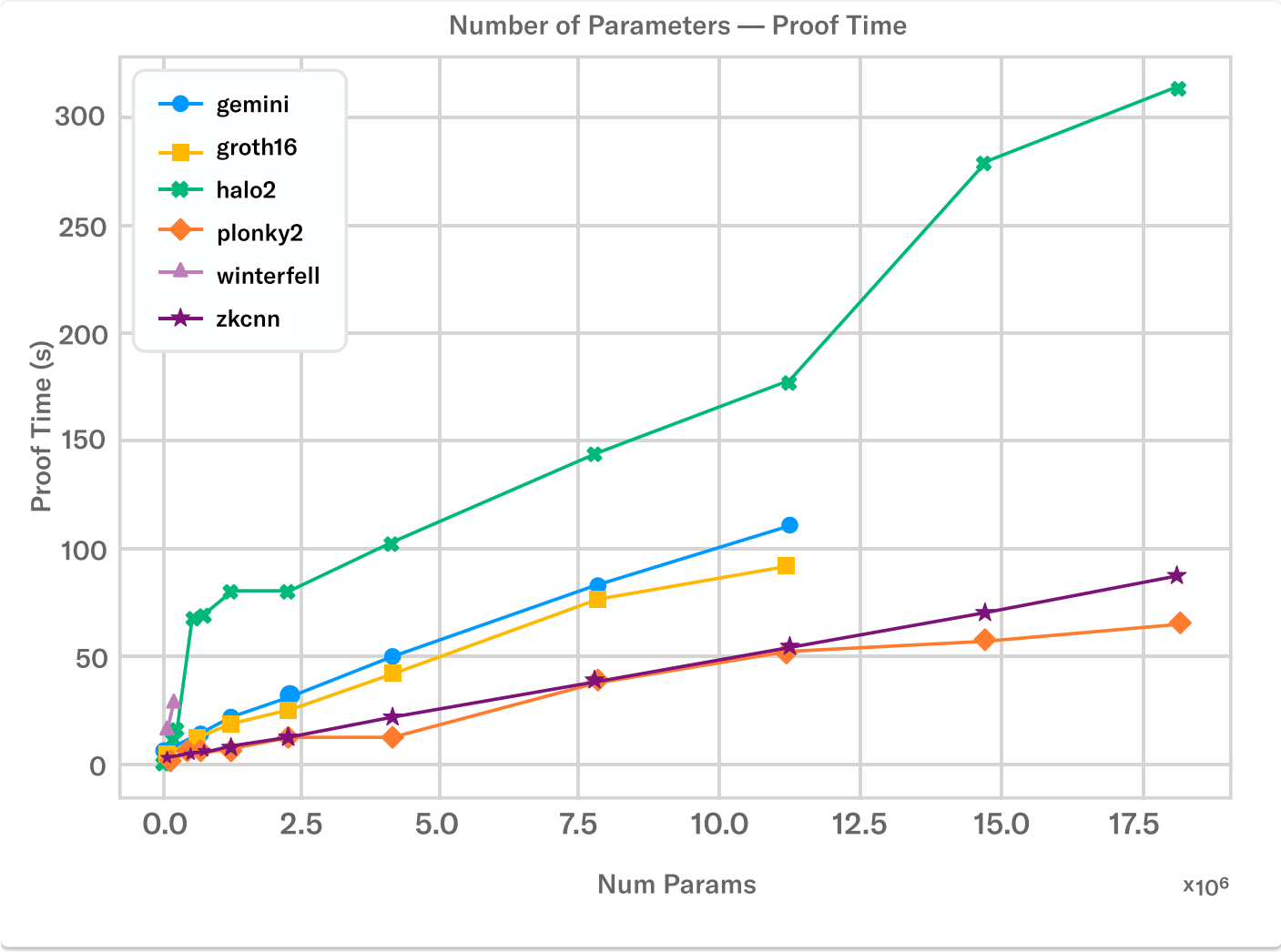


The current state of zero-knowledge systems, even with high-performance hardware, is insufficient to handle large language models. But progress is being made with smaller models.

The Modulus Labs team recently released a paper titled "The Cost of Intelligence", where

they benchmark existing ZK proof systems against a wide range of models of different sizes. It is currently possible to create proofs for models of around 18M parameters in about 50 seconds running on a powerful AWS machine using a proving system like plonky2.

The following illustrates the scaling behaviour of different proving systems as the number of parameters of a neural network are increased:"

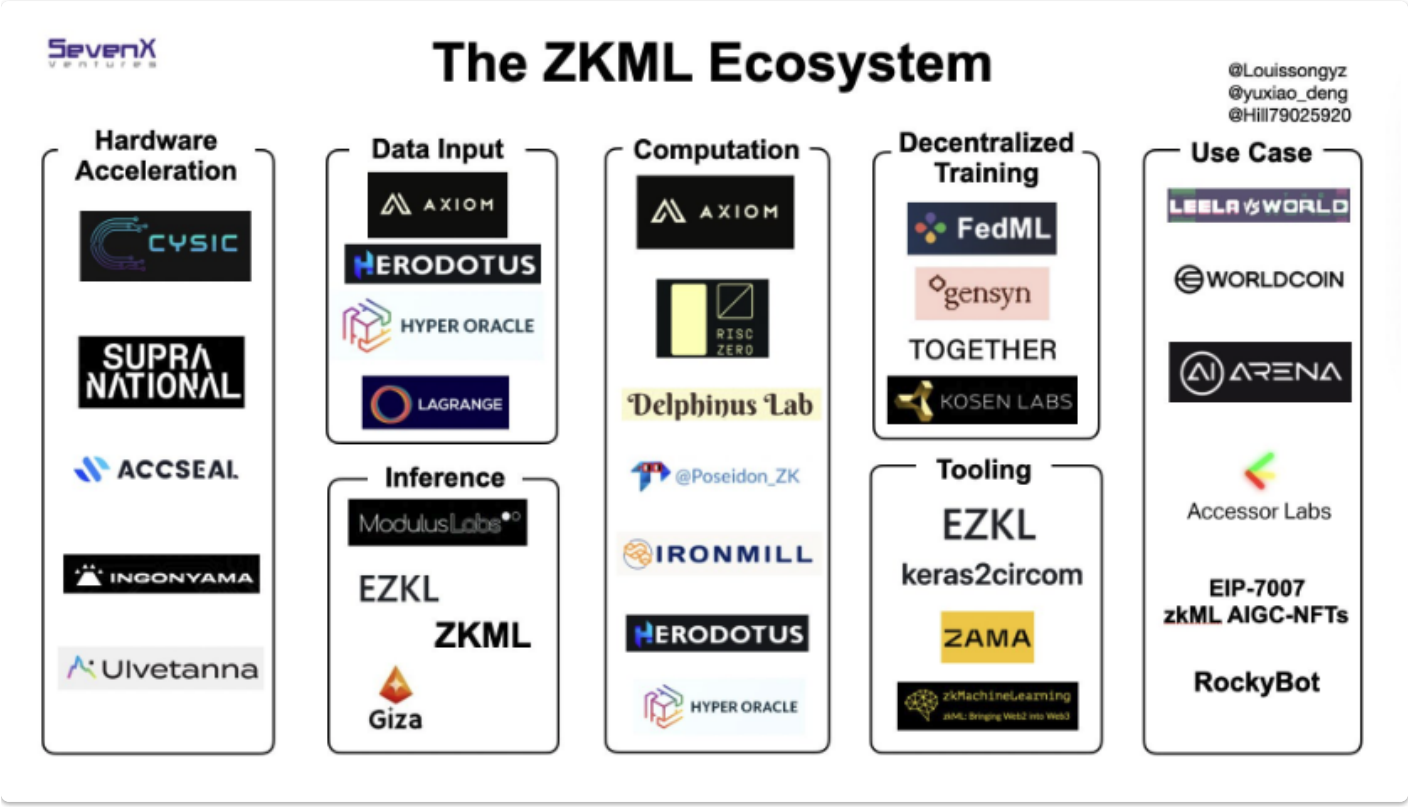


Modulus Labs

Modulus Labs are working on a number of use cases :

- On-chain verifiable ML trading bot - RockyBot
- Blockchains that self-improve vision
- Enhancing the Lyra finance options protocol AMM with intelligent features
- Creating a transparent AI-based reputation system for Astraly (ZK oracle)

Ecosystem


SevenX
The ZKML Ecosystem
@Louissongyz
@yuxiao_deng
@Hill79025920
Hardware Acceleration
CYSIC
SUPRA NATIONAL
ACCSEAL
INGONYAMA
Ulvetanna
Data Input
AXIOM
HERODOTUS
HYPER ORACLE
LAGRANGE
Inference
ModulusLabs
EZKL
ZKML
Giza
Computation
AXIOM
RISC ZERO
Delphinus Lab
@Poseidon_ZK
IRONMILL
HERODOTUS
HYPER ORACLE
Decentralized Training
FedML
gensyn
TOGETHER
KOSEN LABS
Tooling
EZKL
keras2circom
ZAMA
zkMachineLearning
Use Case
LEELA vs WORLD
WORLDCOIN
AI ARENA
Accessor Labs
EIP-7007
zkML AIGC-NFTs
RockyBot

Approaches to modelling a neural network

From article by Pratyush Ranjan Tiwari

1. Naive implementation using addition/multiplication gates: The circuit would be very simple and any proof protocol could be utilised but the size of the circuit is very large $O(n^2 \cdot w^2)$ for image matrix being $n \times n$ and the weight matrix being $w \times w$. Making this method inefficient for any powerful CNN with more than a couple layers.
2. Compute convolutions using FFT: This was demonstrated by an S&P (Oakland) '20 paper ZXZS20 and results in a circuit of size $O(n^2 \log n)$ with $O(\log n)$ depth. Another variant of this approach was followed by the zkCNN paper by utilizing a sumcheck protocol for FFT, which is asymptotically optimal with prover time $O(n^2)$.
3. Convolution $\approx$ Polynomial multiplications: This approach was demonstrated by the vCNN paper. It uses the elegant approach

of first computing the result of the convolution outside the circuit and then given the result as input, it checks equality of polynomial multiplication.
The equality check is done by checking equality and random points, the security comes from the Schwartz-Zippel Lemma
This is the most efficient approach as the zk proof circuit evaluating polynomials at random points is of size $O(n^2 + w^2)$.

A major problem with these approaches is the need to convert floating point numbers to integers for the proofs.
Similarly efficiently proving the correctness of matrix multiplication is difficult.

Some possible workarounds are 'quantise' the weights so that they can be represented as elements in a finite field. This still has the problem of the modular nature of the field.

Tensor Plonk

See post by Daniel Kang
TensorPlonk is a described as a “GPU” for ZKML.
TensorPlonk can deliver up to 1,000x speedups for certain classes of models, such as the Twitter system above.
Using TensorPlonk, the proving cost for the Tweet example above would be ~$30 compared to ~$88,704.

	TensorPlonk	ezkl (no hash)	ezkl (with hash)
Proving Time	6.7 seconds	1517.5 seconds	> 6 hours*
Verification Time	70 ms	250 ms	> 250 ms*
Proof Size	12.5 kb	88 kb	~800 kb*

TensorPlonk optimizes matrix multiplications, non-linearities, and weight commitments with new optimizations and by extending recent work.

The second part of TensorPlonk accelerates the non-linearities.
To understand why we need this, existing work in ZKML that uses lookups uses a lookup proving argument plookup.

The proof is split into a “circuit” and “lookup tables.”
With plookup, the circuit must be the size of the table, so if we use a large table, we must use a large circuit.
With our optimizations, the lookup table can be as large as 2^{24} elements but the circuit can be as small as 2^{16} elements! This means we would have an extra 250x overhead to use a large lookup table .

Instead, we can leverage cached quotients : cq, which allows for lookup tables of size independent of the circuit size.
Tensor Plonk further optimised cq to “batch” work across lookup tables, which can reduce the computations for the lookups by up to 3x.

The third optimisation optimises the weight commitments, which binds the model provider to use a fixed set of weights.
Instead of using a hash, which can be extremely expensive, we use a KZG commitment of the weight vectors which can reduce the proving time by up to 10x.

Weightless networks

See article

An approach from the zero gravity team winners of the ZK Lisbon hackathon '23 was use a neural net design that didn't use weights. From their description

"We propose a different approach: let’s go back to a time before the NN paradigm was settled, to a time when a greater variety of neural nets roamed the earth, and let’s find a machine learning model more amenable to ZKP. One such model is the “Weightless Neural Network”.

Weightless means no weights, no floating point arithmetic, and no expensive linear algebra, let alone non-linearities

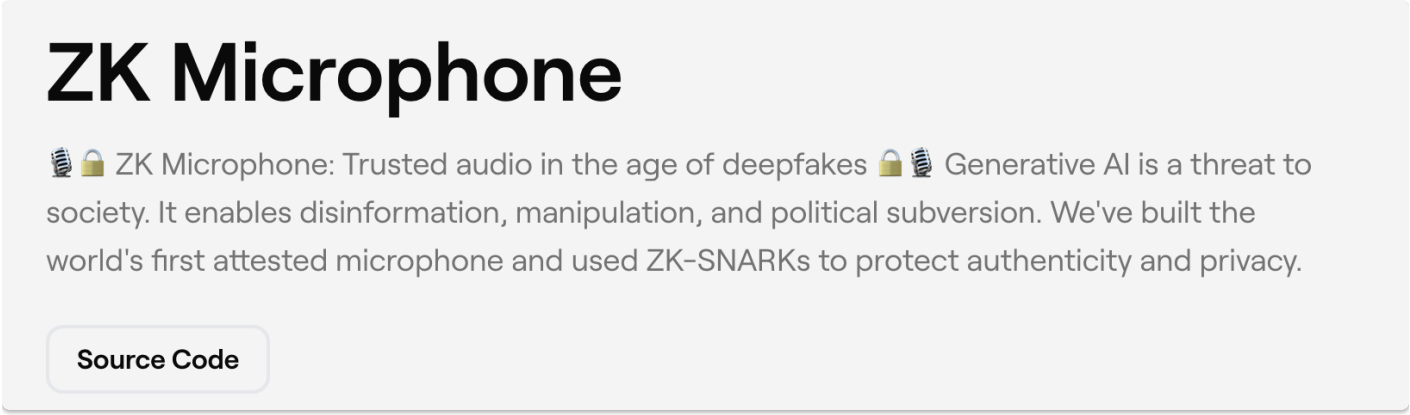
A Weightless Neural Network (WNN) is entirely combinatorial. Its input is a bitstring (e.g. encoding an image), and their output is one of several predefined classes, e.g. corresponding to the ten digits. It learns from a dataset of (input, output) pairs by remembering observed bitstring patterns in a bunch of devices called RAM cells, grouped into “discriminators” that

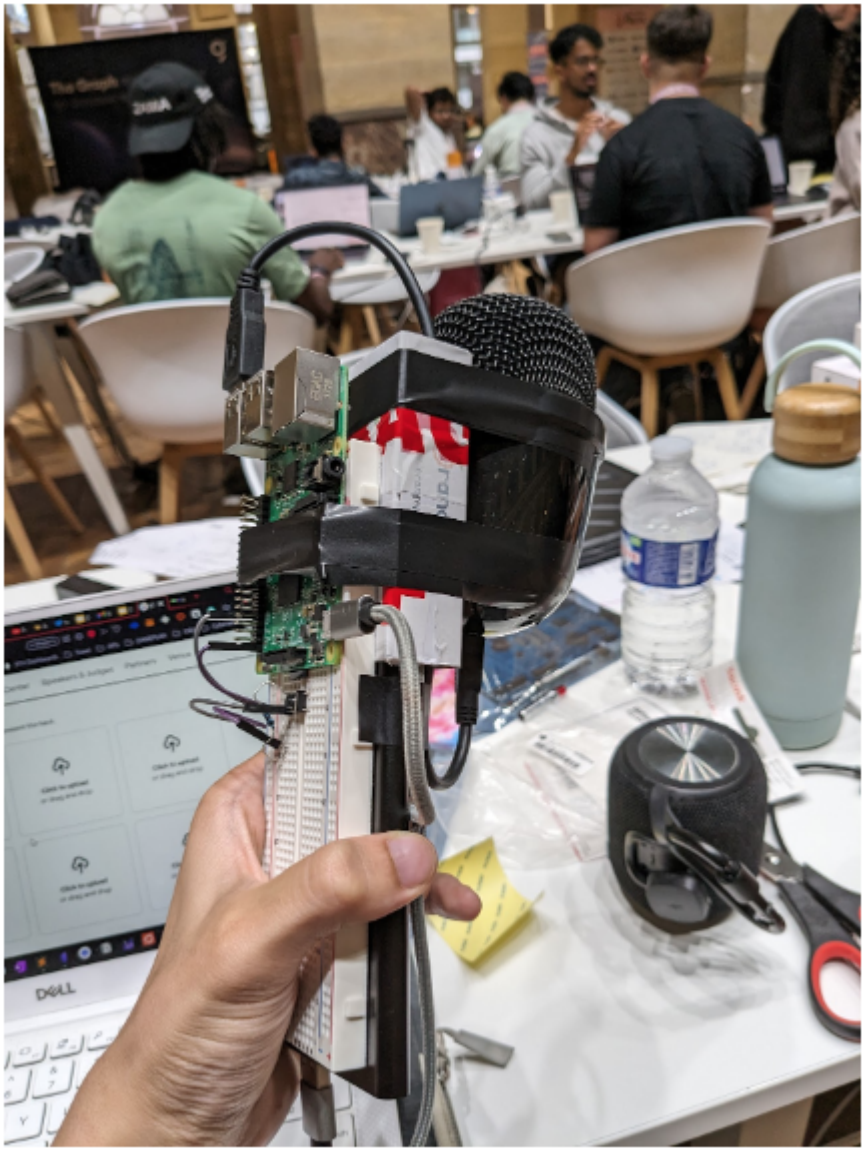
correspond to each output class. RAM cells are so called since they are really just big lookup tables, indexed by bitstring patterns, and storing a 1 when that pattern has been observed in an input string that is labeled with the class of this discriminator.

Zero Gravity is a system for proving an inference run (i.e. a classification) for a pre-trained, public WNN and a private input. In Zero Gravity, the prover claims to know an input bitstring x such that the public model classifies it as class y. The input x can be treated as a private input, in which case the system is zero-knowledge: although inference does reveal something about x to the verifier (namely its corresponding output class y), this information is already contained in the statement being proved.

Applications

Daniel Kang is working on verifying image authenticity.
A similar project is the ZK Microphone.
See recent project from ETH Global

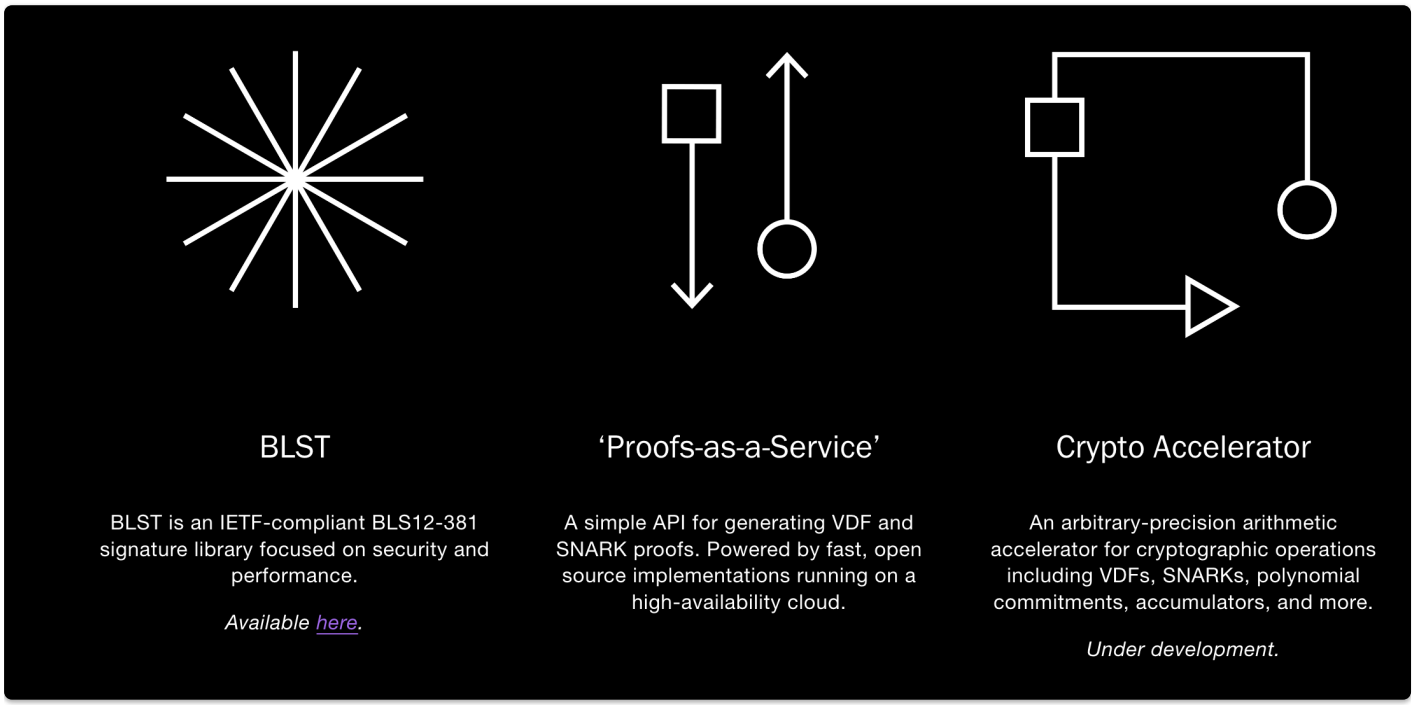




Also see this talk

Supranational

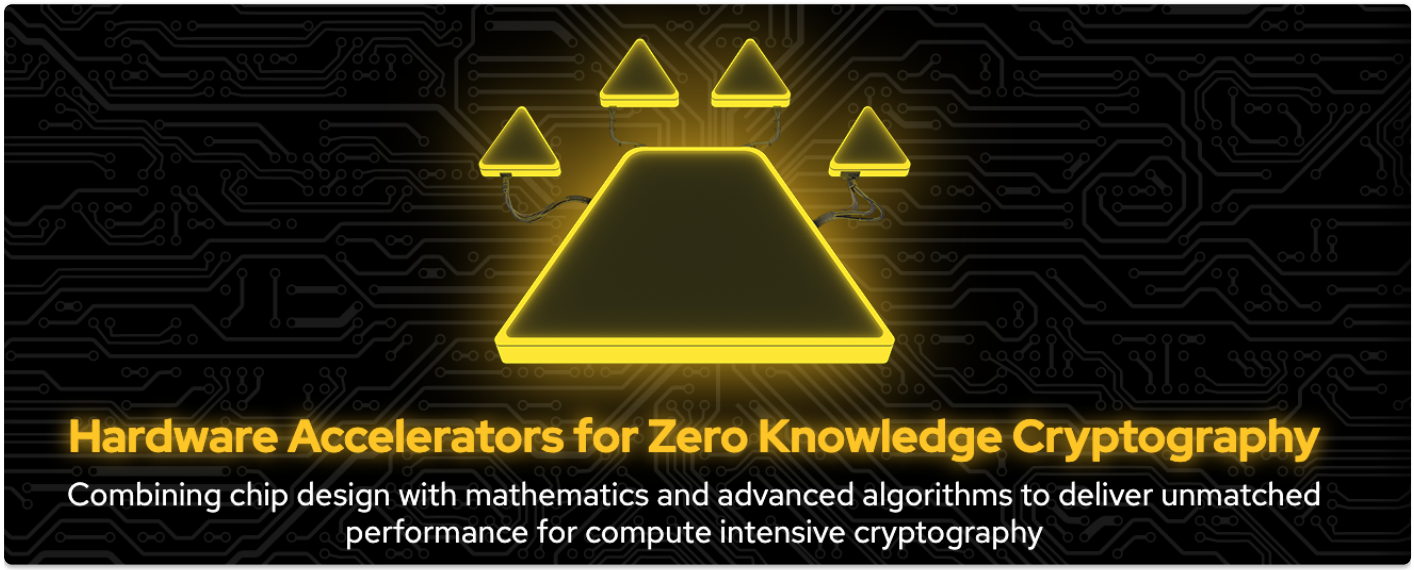
See site



Also see VDF day videos

Ingonyama - Hardware for zkML

See Site

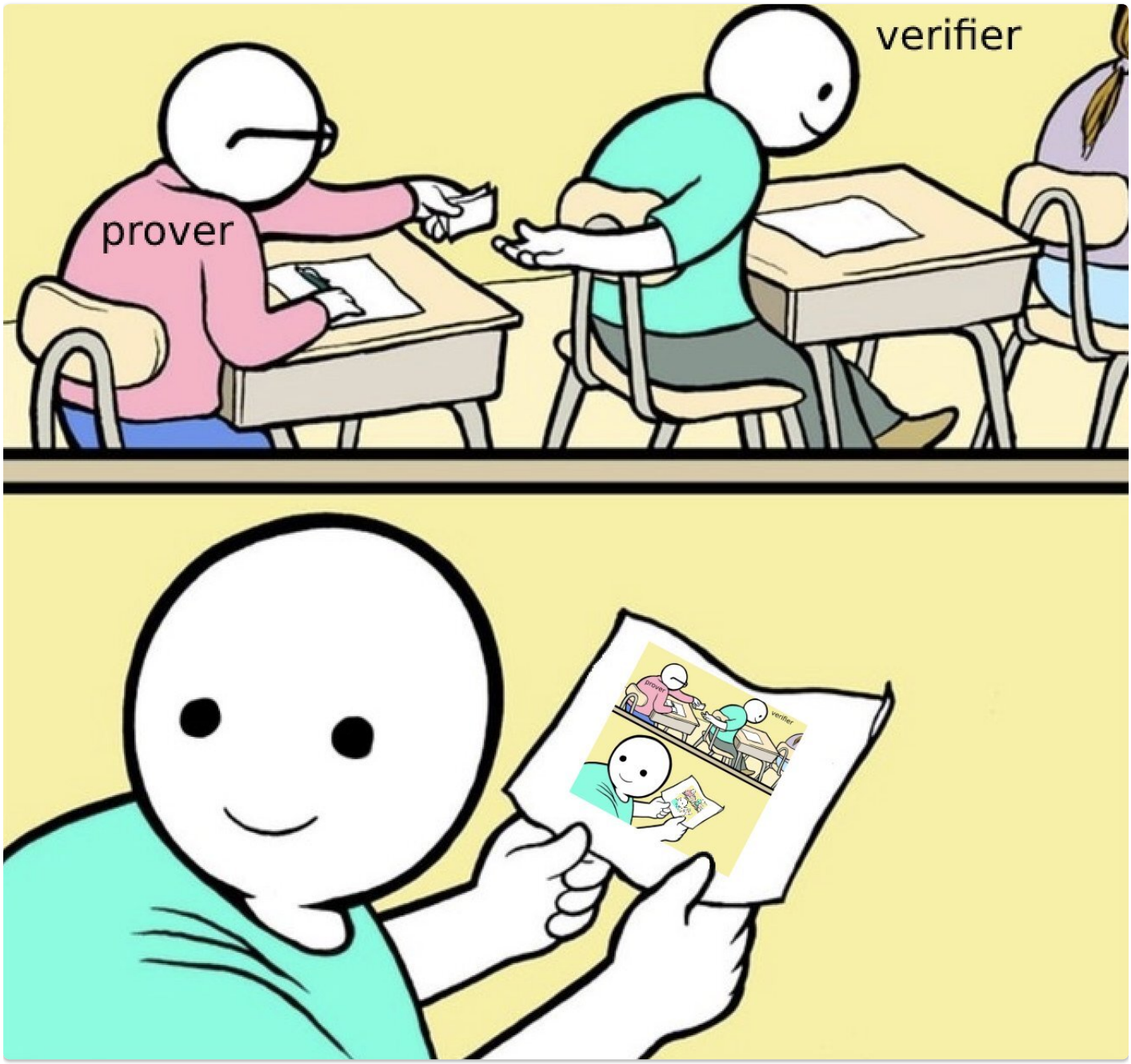


See this talk about the 'ZPU'

zkML Resources

dcbuilder has many resources
Introduction to zkML from Worldcoin
Worldcoin awesome repo
Zero Knowledge Podcast
Image redacting using zkSNARKS paper
ZK ML community [calls](- ZKML community call #0) on telegram
Research from PSE team at EF

Recursion overview



The ability to create recursive proofs is a very powerful technique.
Proof recursion is at the heart of the Mina blockchain - a succinct blockchain.

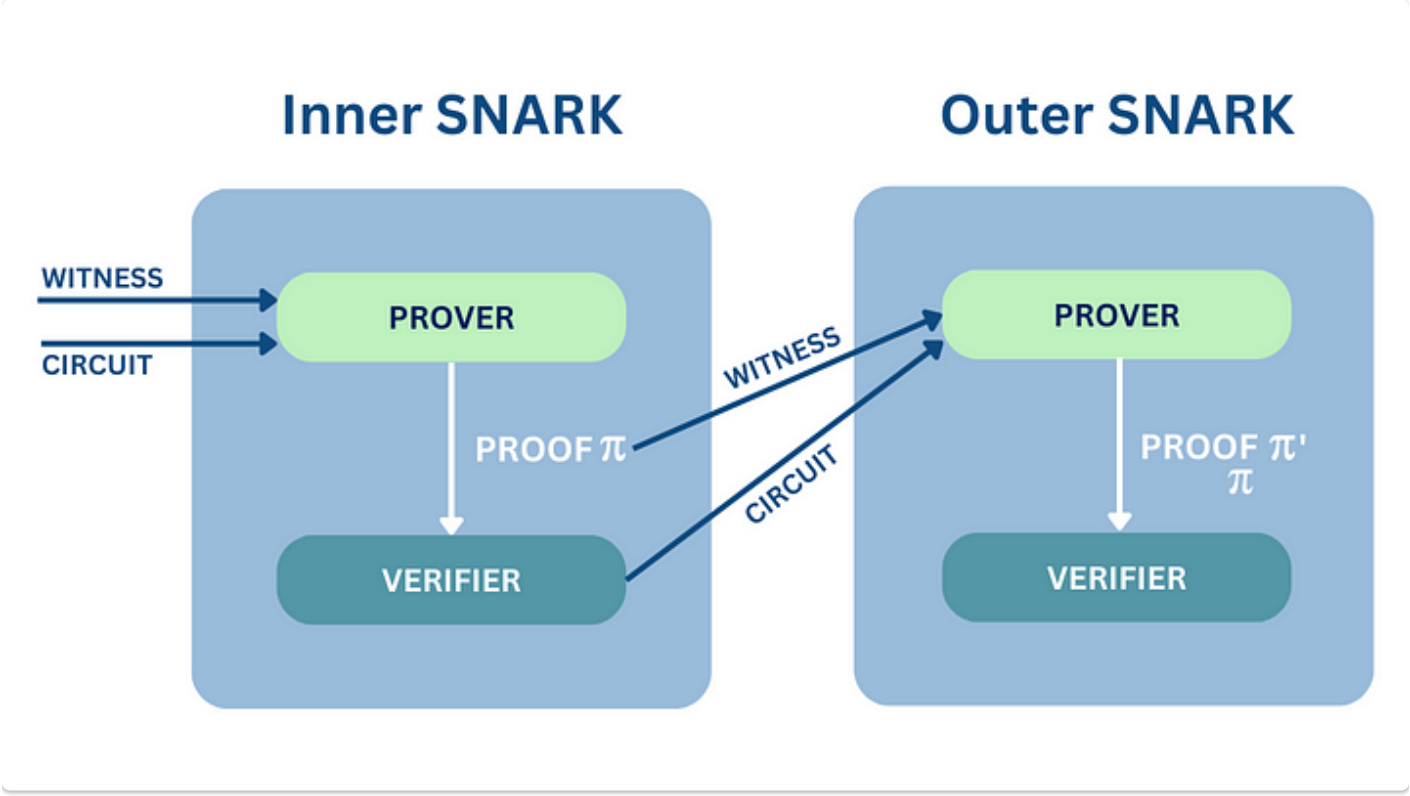
What we are trying to achieve using a common circuit is

- Step 1 : have proof A as an input to the circuit that will produce proof B
- Step 2 : Proof B is the input to proof C ...

The how of recursion

See video from Stanford MOOC

One approach is to include a verifier within your circuit so that the verification of a previous proof can be part of the current proof.
From Article



A problem with this approach is if the verifier becomes too large, then the recursion will 'blow up' and become infeasible.

To do this we use cycles of elliptic curves. The curves are specially chosen to have

complementary properties.

These curves are referred to as “tick” and “tock” within the Mina source code.

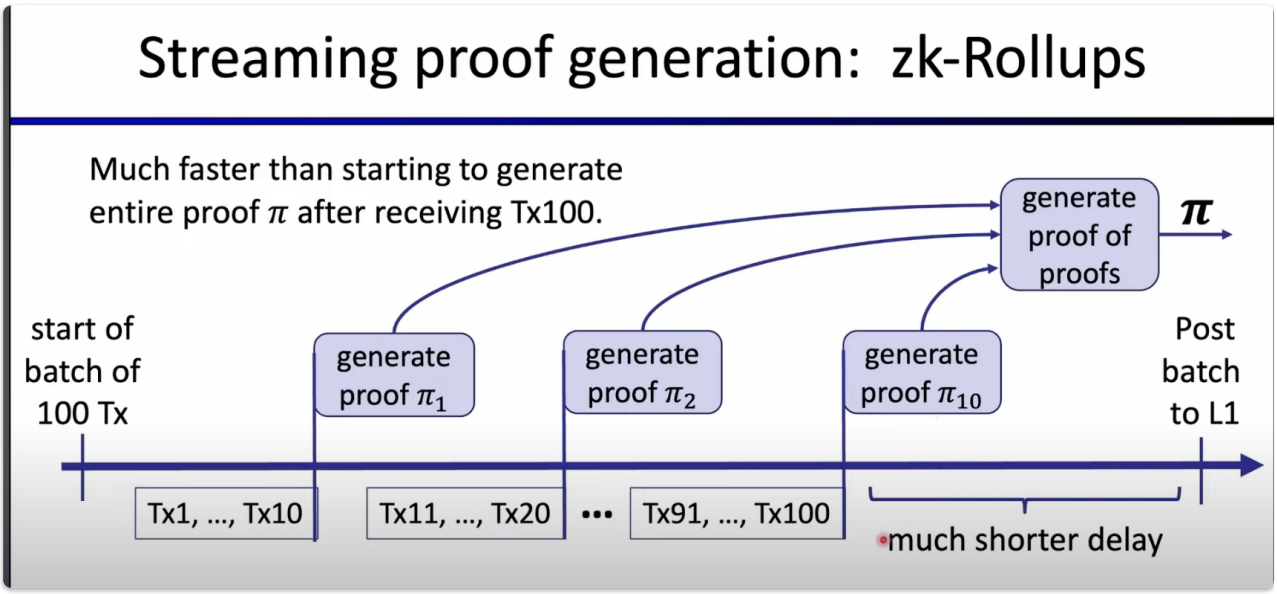
Tock is used to prove the verification of a Tick proof and outputs a Tick proof.
Tick is used to prove the verification of a Tock proof and outputs a Tock proof.
In other words,

- Provetock(Verify(Tick))=Tickproof
- Provetick(Verify(Tock))=Tockproof

Recursive proofs

Applications of recursion

- Proof compression
    - for fast prover and short proof
- Streaming Proof generation
  - as in a rollup for example



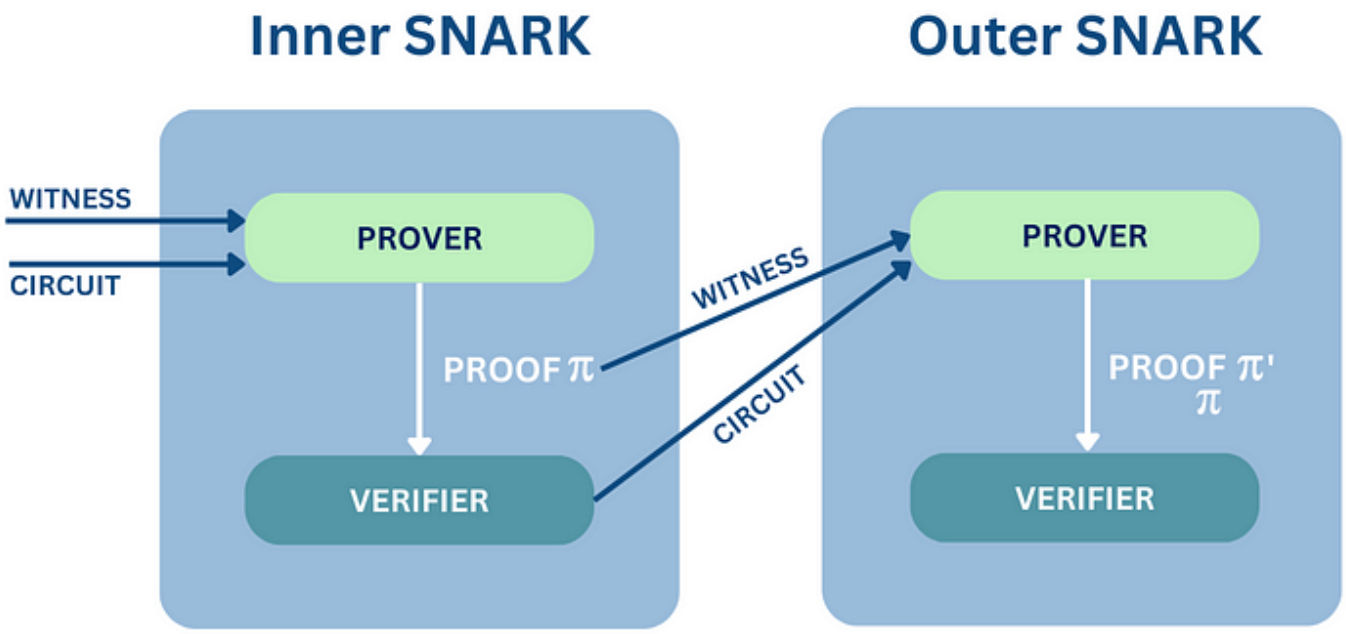
- IVC (See Valiant 08)
  See original paper
  See review

Approaches to recursion

See video from Stanford MOOC

One approach is to include a verifier within your circuit so that the verification of a previous proof can be part of the current proof.

From Article



A problem with this approach is if the verifier becomes too large, then the recursion will 'blow up' and become infeasible.

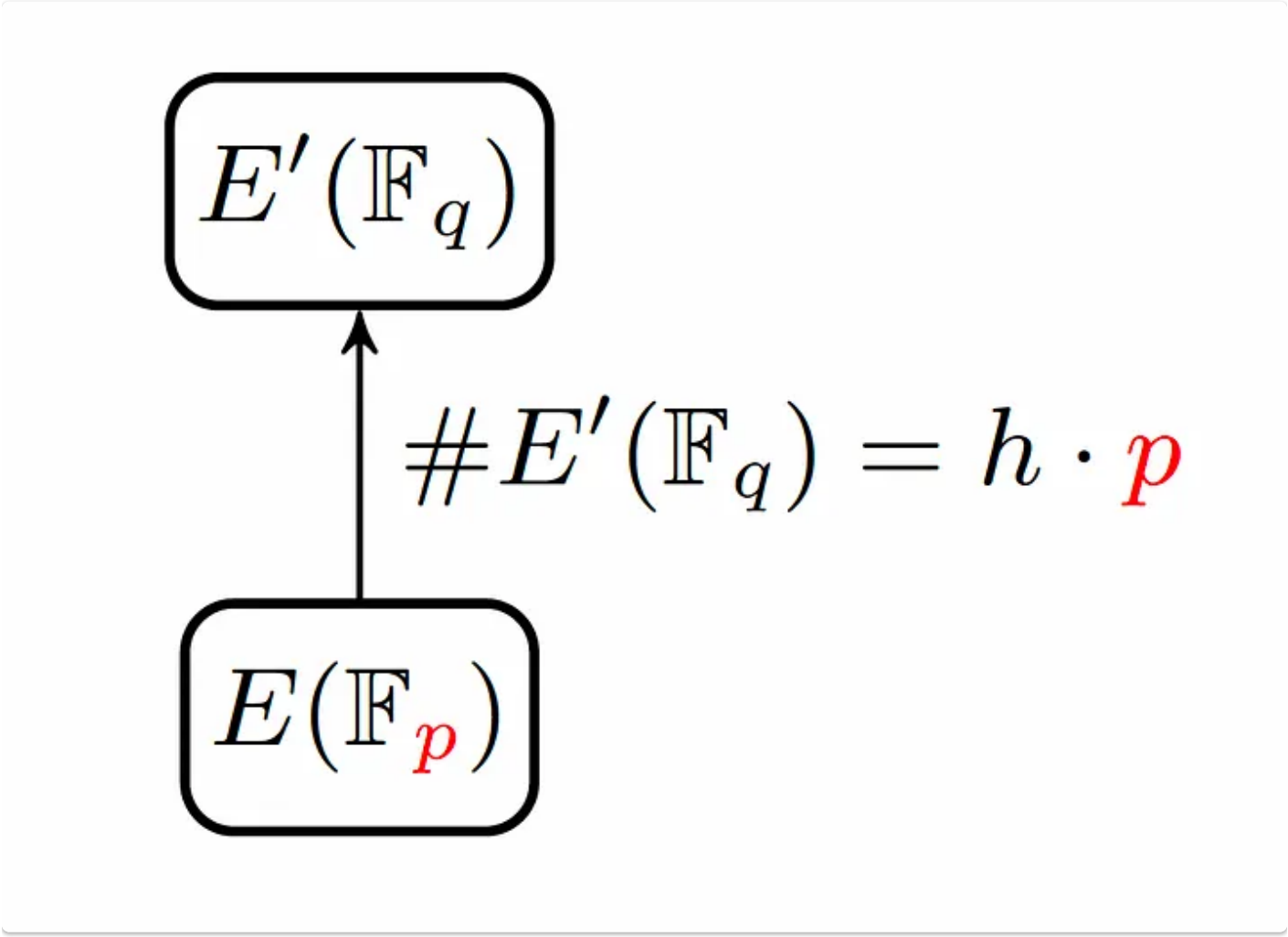
Cycles of curves

At the cryptographic level, recursion can be achieved by using cycles of curves, the curves have to be chosen so that the scalar field of one curve has the same order as the base field of the other curve.
SNARK circuit elements live in the curve's scalar field, while the proof is encoded in the base field

Background

An elliptic curve A is defined over a field $\mathbb{F}_q$. The resulting elliptic curve subgroup has order p. The cofactor is defined q/p

If the cofactor of A is 1, there will always exist a corresponding curve B defined over $\mathbb{F}_p$, whose subgroup has order q.
A SNARK circuit defined over curve A can efficiently evaluate group operations over curve B and vice-versa.



A 2 chain cycle as used by Celo and Aleo

Drawbacks of curve cycles

From Field selection article

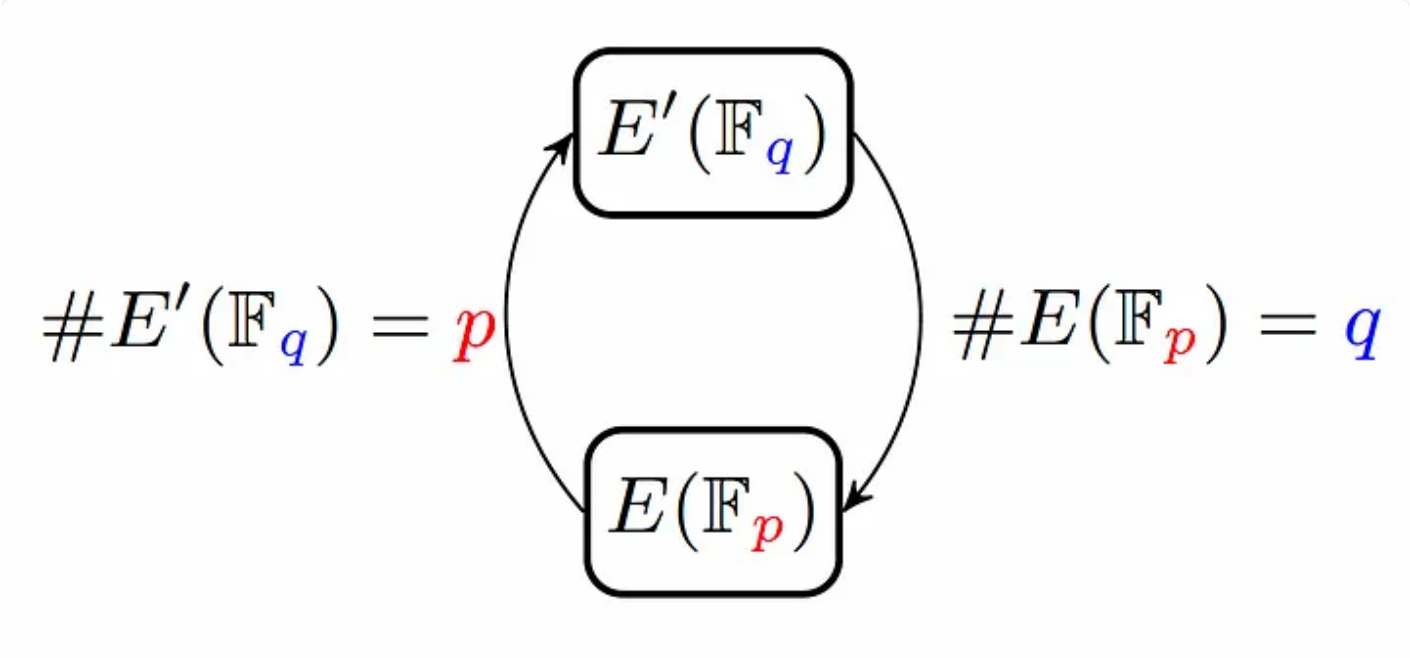
"The Polynomial Commitment Scheme (PCS) part in non-pairing based schemes has a linear verification time, which will proportionally increase proof verifications cost and circuit size.

The additional time complexity may be amortized with accumulation techniques.

Examples of non-pairing friendly or hybrid cycles include ZCash's Pasta Curves (also used in Mina).

In a nutshell, the idea is to enable proof recursion (proof verifying proofs) and to delay the linear part of the computation to the last layer of recursion."

A hybrid cycle



Taken from the Goblin Plonk documentation (see below)
Verification of a BN254 proof inside a BN254 circuit requires non-native group operations.

One solution is to define a Grumpkin circuit that performs BN254 scalar multiplications.

The BN254 circuit can now verify the correctness of this Grumpkin proof instead of directly performing non-native group operations.

However this approach is not perfect. One must now verify 2 proofs instead of 1 (assuming one is evaluating core protocol logic over 1 curve and uses the cycle curve purely for recursive verification).
Each proof requires a nontrivial amount of hashing, particularly if the proof system's IPA uses a sumcheck protocol .

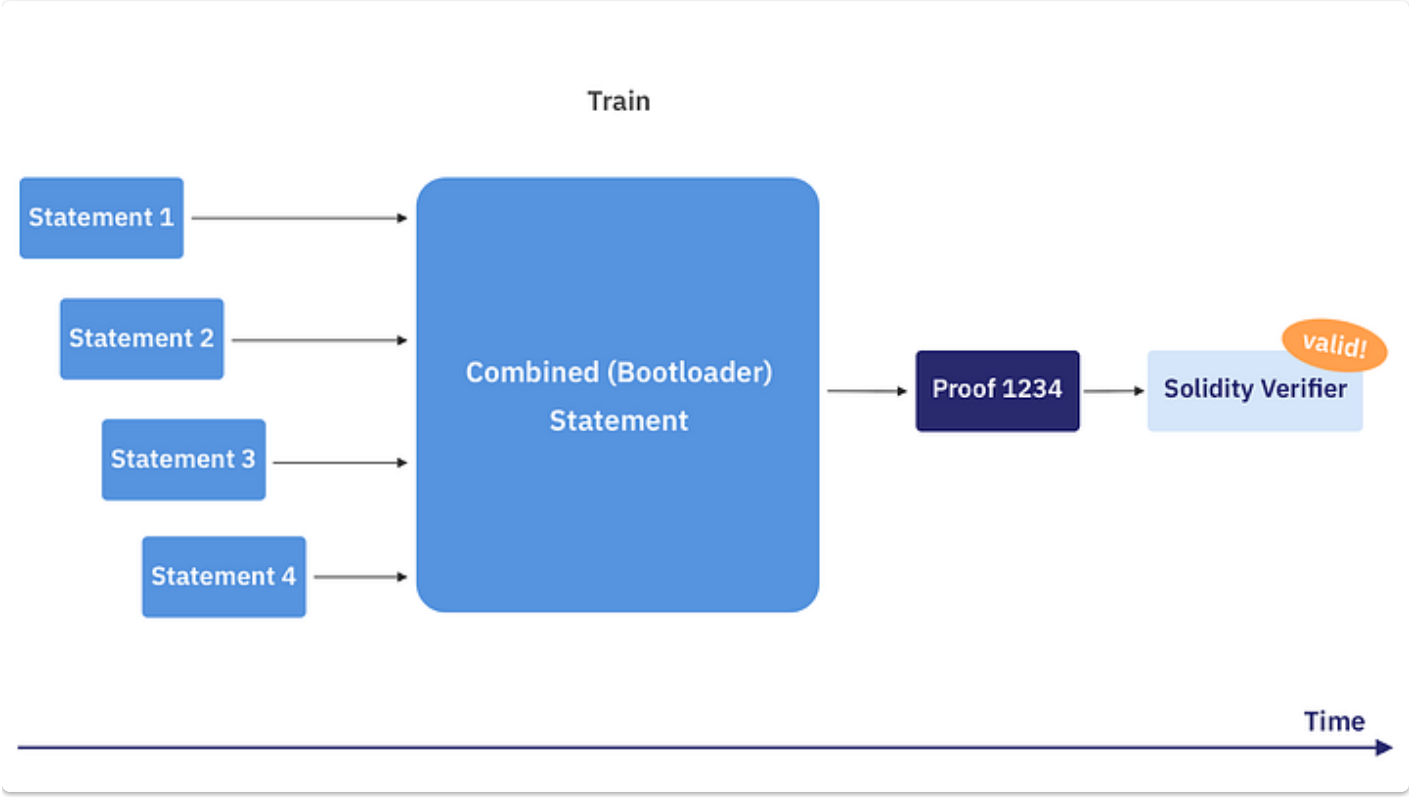
In addition, the *field* operations required to verify the Grumpkin proof are non-native operations in the BN254 circuit.

It is a non-trivial task to delegate these field operations to a future Grumpkin circuit in the

recursion stack. At some point we must link curve A circuit witnesses to public inputs of a SNARK over the cycle curve. This will always require (many) non-native field operations.

Recursive STARKS on Starknet

The original prover was non recursive



From article

Statements arrive over time.

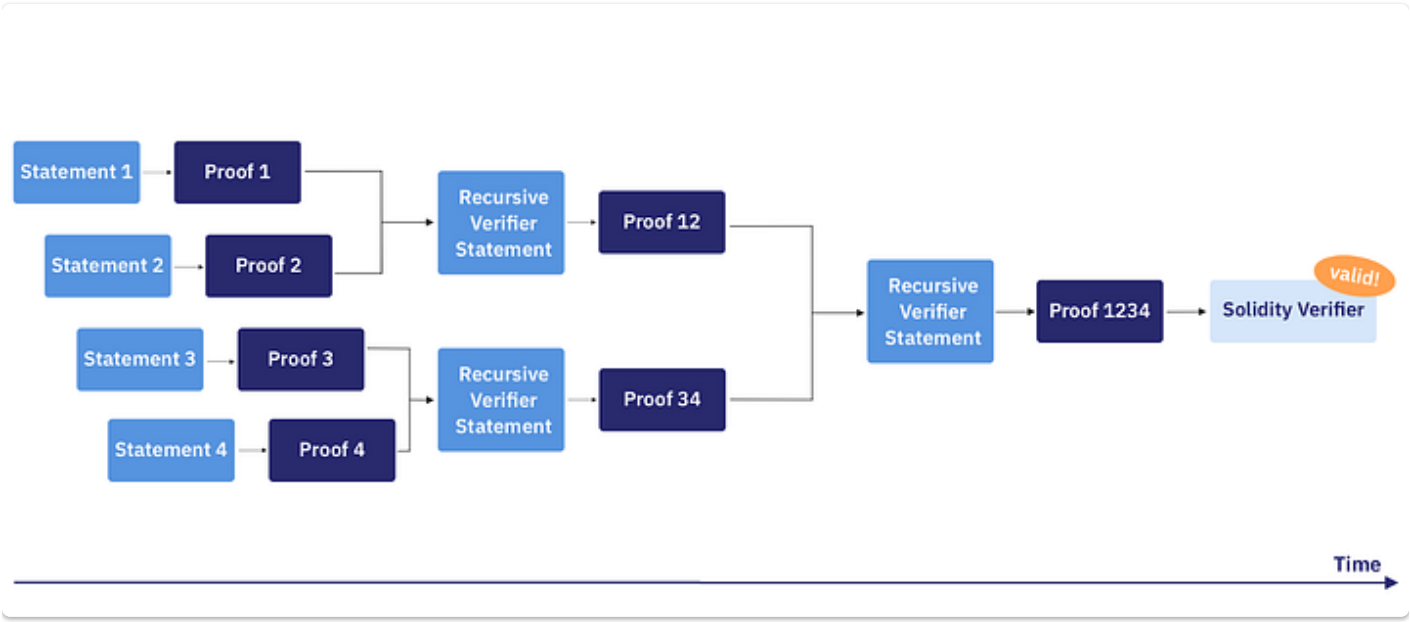
When a certain capacity (or time) threshold is reached, a large combined statement (a.k.a Train) is generated.

This combined statement is proven only once all the individual statements have been received.

This lead to a bottleneck, in particular the amount of memory needed to hold all of the proofs was prohibitive.

Recursive approach with Cairo

A Cairo program can be written that verifies multiple proofs to create a single proof.



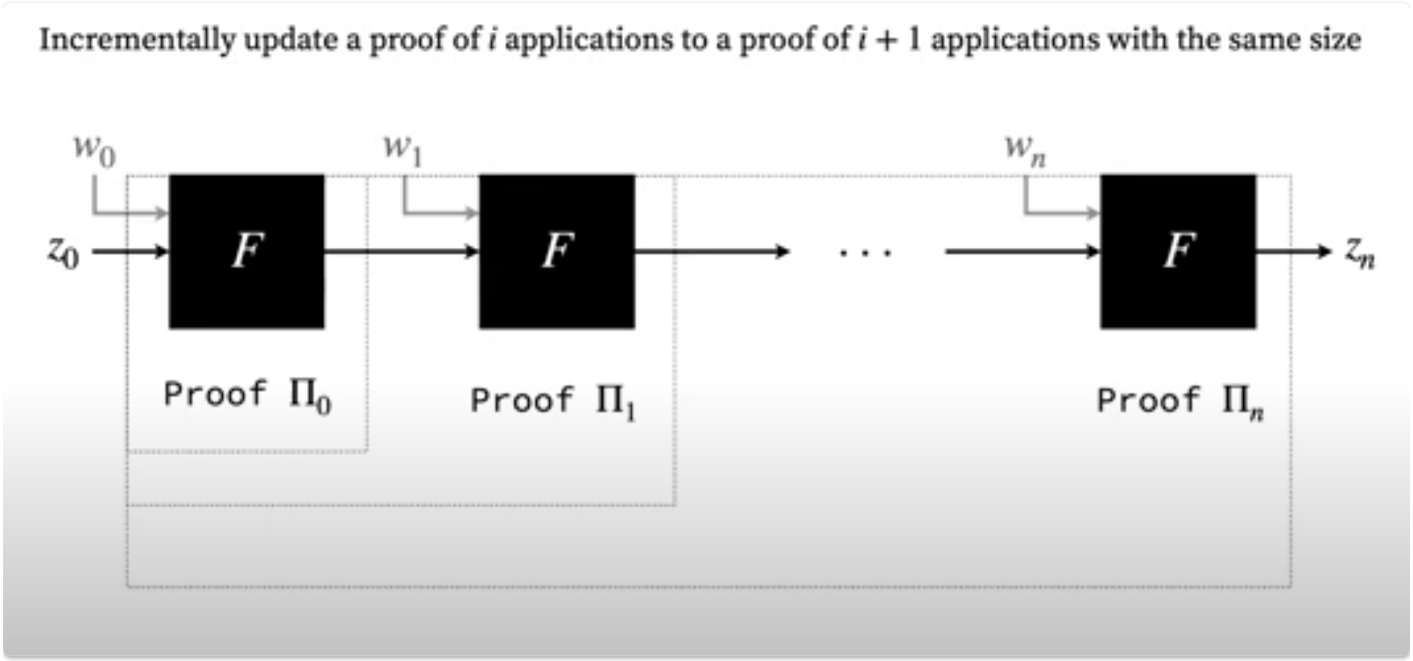
Using recursion in this way reduces the amount of memory needed and means that the whole process doesn't need to wait for all the statements to arrive before it can start.

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



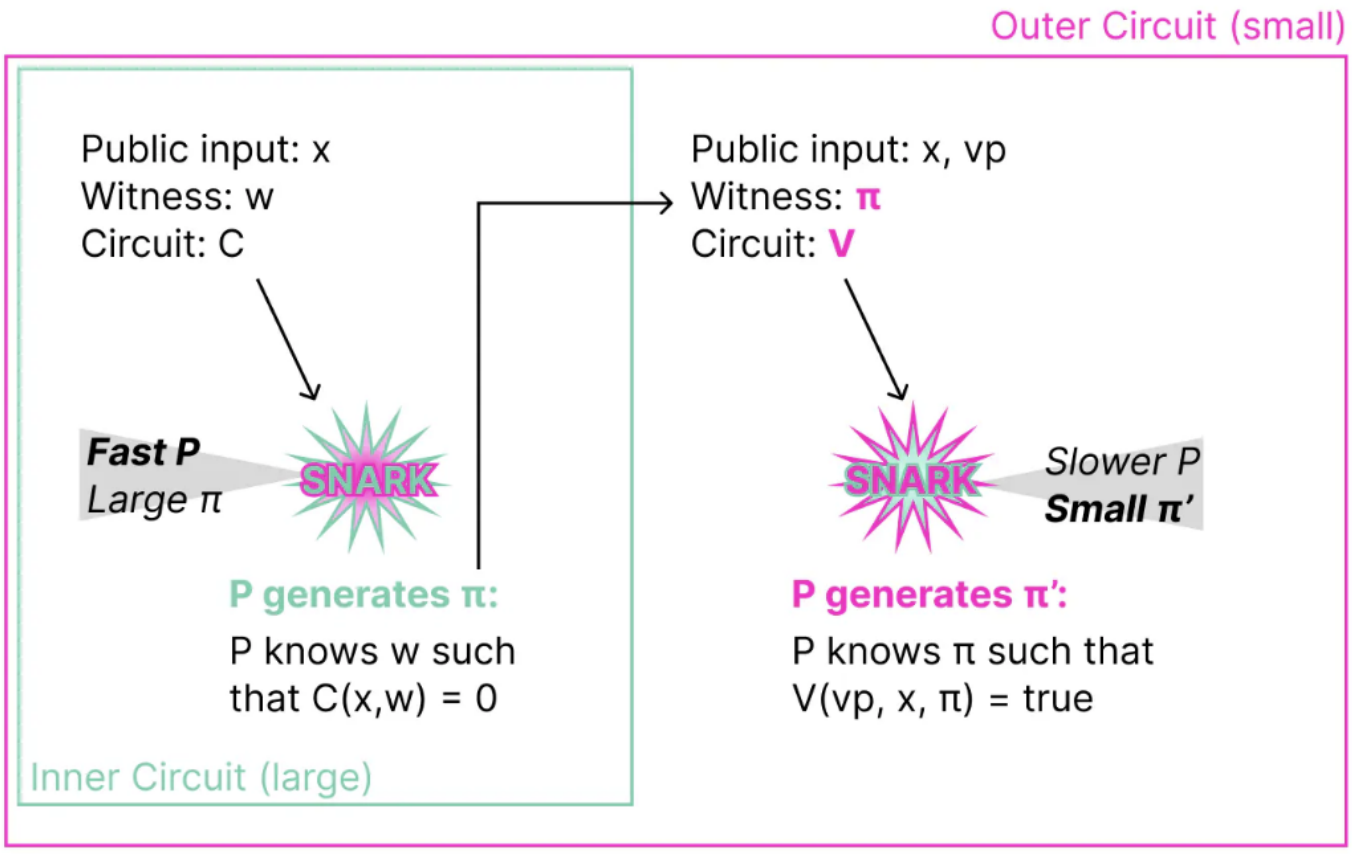
Recursion and aggregation

From Taiko article

Circuits - inner and outer

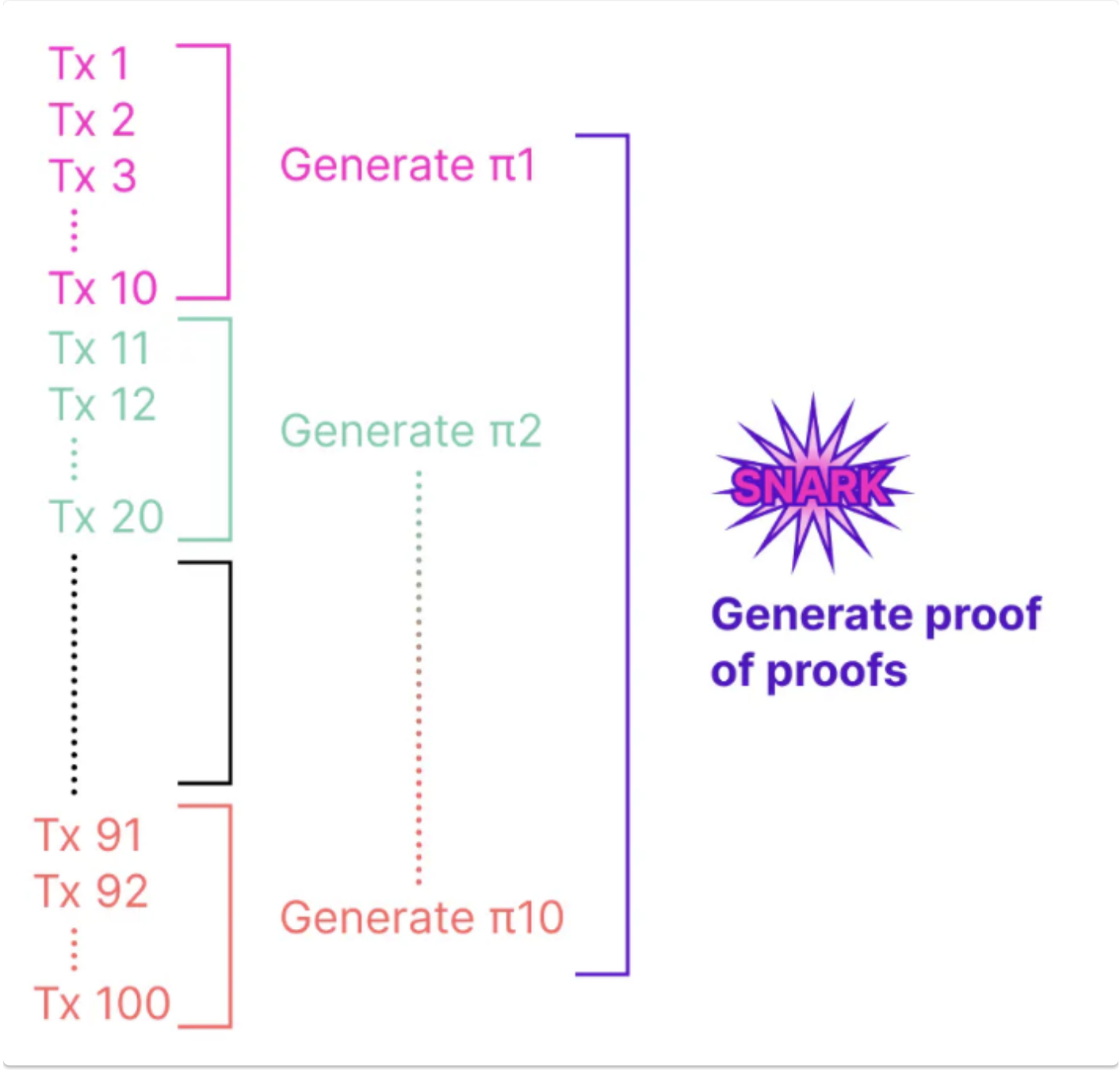
1. Inner Circuit (large): Prover proves that they know the witness. At this stage, proof generation is fast, but the proof is large (except for the case when the proof size is a constant);
2. Outer Circuit (small): Prover proves that they know the proof. At this stage, proof generation is slower (but it's not crucial as in most cases the circuit is much smaller than the first), but the proof is small.

P – Prover
V - Verifier
π – proof (SNARK)
vp – public parameters for V

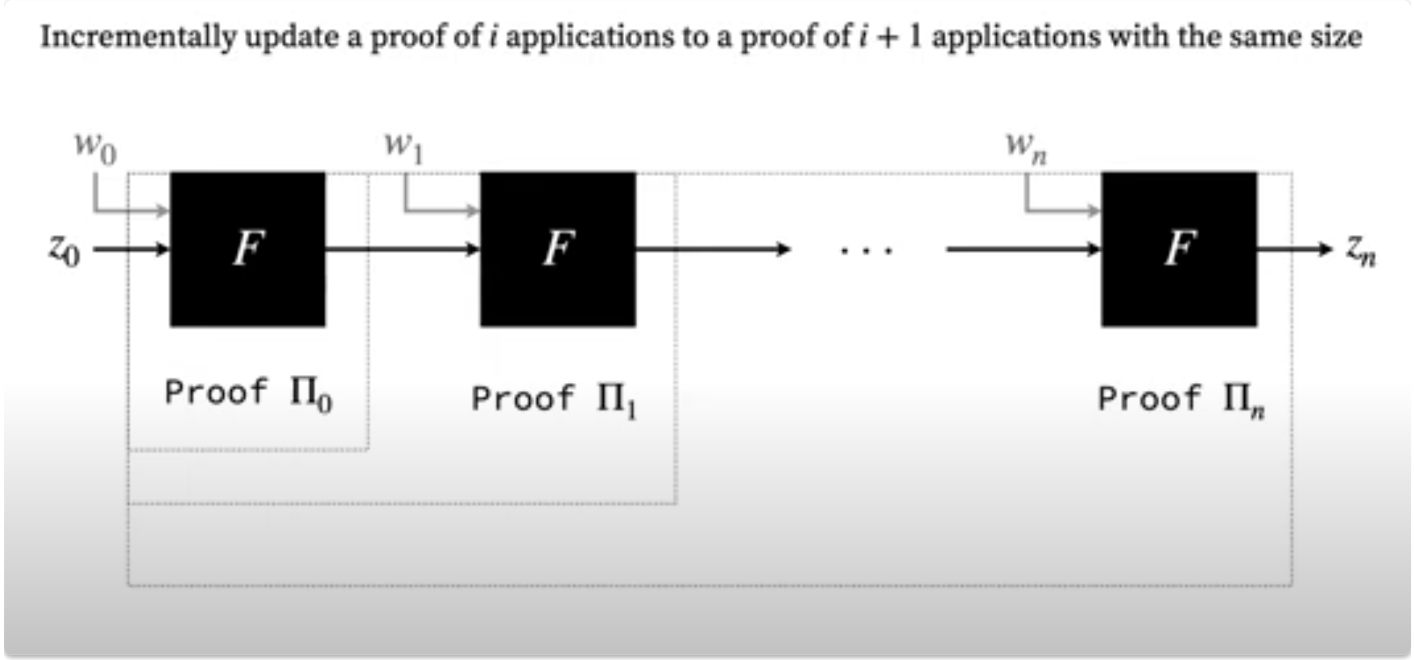


Proof aggregation

For example, as done in zk rollups



IVC - incrementally update proofs

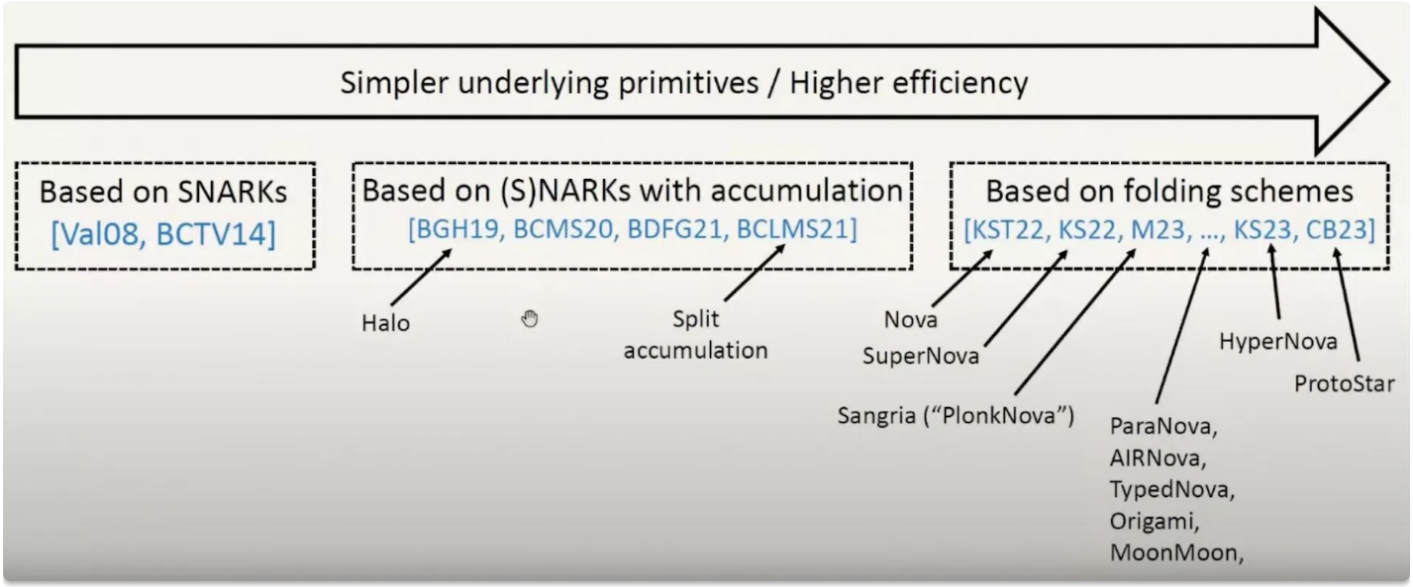


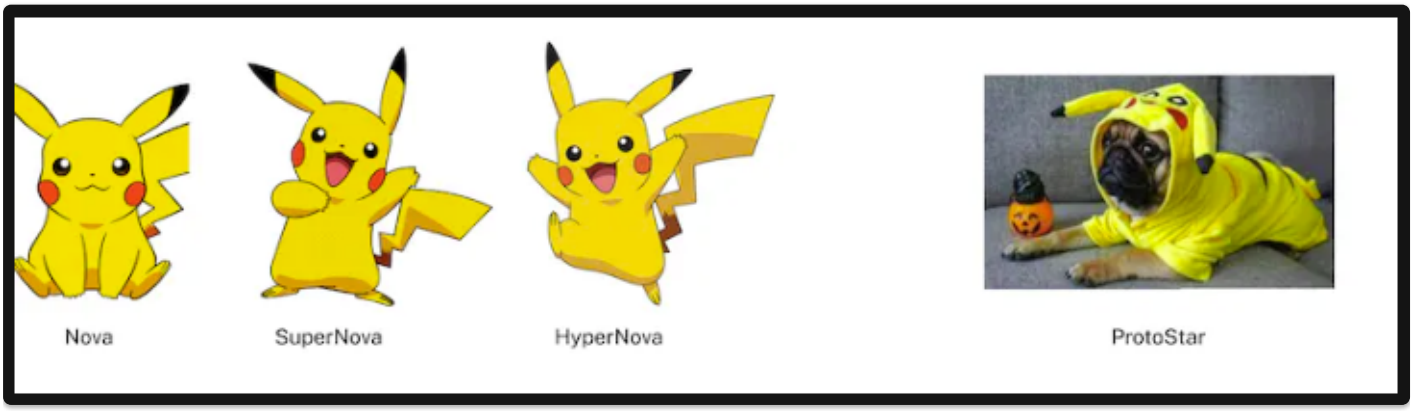
Folding instead of recursion

Advantages of folding for a function F that we are iterating over.

- Doesn't perform any polynomial divisions or multiplications (ex., via FFTs that require much memory and are computationally heavy);
- Works with *any* type of elliptic curves (ex., secp, secq, Pasta, etc.);
- F is specified with R1CS;
- No trusted setup;
- Compared to the recursion overhead, folding is expected to be 5-50x faster than Groth16, PLONK, Halo, etc. Prover time is dominated by two multi-exponentiations of size O(C), where C = |*F*|;
- Verifier Circuit is const-size (two scalar multiplications);
- Proof size is O(log C) group elements (few KBs).

Main Projects





NOVA

See Paper

See article for an in depth description.

See Video from Justin Drake

If our constraints were all linear, folding would be simpler, however R1CS is not linear.
Nova introduced the idea of 'relaxed' R1CS, by adding additional parameters to the instance.
With this new form, we can take linear

combinations of the instances, to allow them to be folded together.

Sangria

See paper
Nova allowed folding for R1CS arithmetisation, Sangria provides folding for PLONK. Again this requires us to 'relax' the gate constraints by adding extra terms. Fortunately copy constraints do not need to be changed.

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 (these are the terms that are handled by the 'relaxed' format of

arithmetisation), 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

See paper
"A distinguishing aspect of HyperNova is that the prover’s cost at each step is dominated by a single multi-scalar multiplication (MSM) of size equal to the number of variables in the constraint system, which is optimal when using an MSM-based commitment scheme."
This is Nova using ccs and also using a VDF

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.

Accumulation

Accumulation is a simple yet powerful primitive that enables incrementally verifiable computation (IVC) without the need for recursive SNARKs

From Protostar paper
"Instead of verifying a SNARK at every step of the computation, we can instead accumulate the SNARK verification check with previous checks.
We define an accumulator such that we can combine a new SNARK and an old accumulator into a new accumulator.
Checking or deciding the new accumulator implies that all previously accumulated SNARKs were valid
Now the recursive statement just needs to ensure the accumulation was performed correctly. Amazingly, this accumulation step can be significantly cheaper than SNARK verification"

> **Resources**
>
> See awesome repo

Small Fields

See this article for a discussion.

When designing ZKP systems, we can choose the fields we use to help optimise the processes.
There are various factors to consider, such as the architecture of the CPU or virtual machine.

There has been a drive to use smaller fields, as these will work better with the common 32 bit or 64 bit architectures.
For example the Goldilocks field used by Plonky2 has a prime

$$p = 2^{64} - 2^{32} + 1$$

whereas Plonky 3 uses

$$p = 2^{31} - 1 + 1$$

If less than 2^{31} this will fit into 32 bit GPUs

Unfortunately other factors also come into play, because we may want to do operations on larger numbers, for example using signature schemes.
KZG also is problematic with fields of this size,

but can make use of extension fields to mitigate problems.