Lesson 5 - zkML timeline / challenges

Week 2 - Deep Dive

Day	Lesson	Topic	Teacher
Monday	No Lesson		
Tuesday	5	zkML timeline / challenges	Extropy / Worldcoin
Wednesday	6	Giza	Giza / Extropy
Thursday	7	Modulus	Modulus

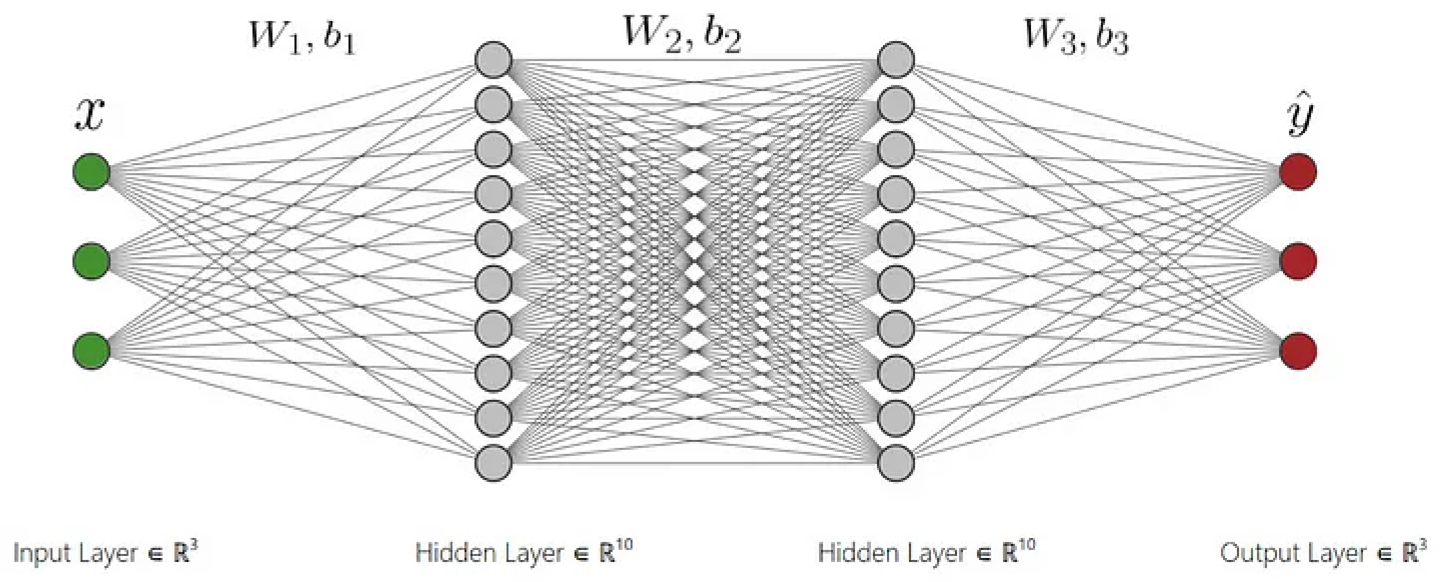
Today's topics

- Review
- Timeline of zkML
- Challenges

Review / points from last week

- We are mainly concerned with proving the inference stage
- Performance is problematic, improvements are likely to come from multiple small improvements. From article

ML is unique because ML computations are generally represented as dense computation graphs that are designed to be run efficiently on GPUs. They are not designed to be proven. So if you want to prove ML computations in a ZK or optimistic environment, they have to be recompiled in a format that makes this possible — which is very complex and expensive.



W_1, b_1
W_2, b_2
W_3, b_3
x
$\hat{y}$
Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^{10}$
Hidden Layer $\in \mathbb{R}^{10}$
Output Layer $\in \mathbb{R}^3$

Terminology

SRS / URS - Structured Reference String / Universal Reference String

When we setup a ZK proving system, in addition to the program we need some data called a reference string, which is used when we are creating proofs. In more recent zkSNARKs we need only create this once and it can be re used if we change our prorgam and have to setup the system again.

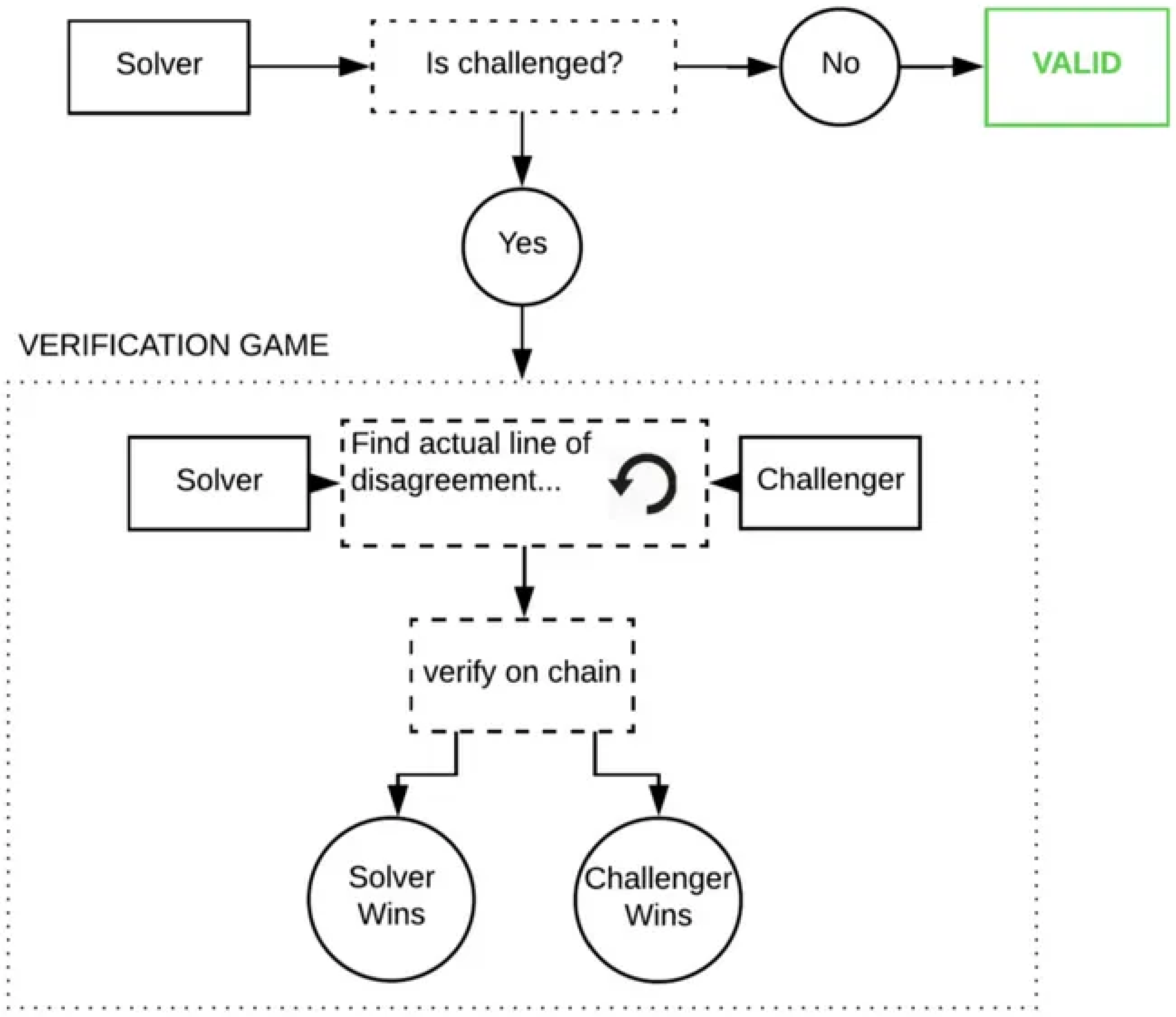
Proving Key / Verification Key

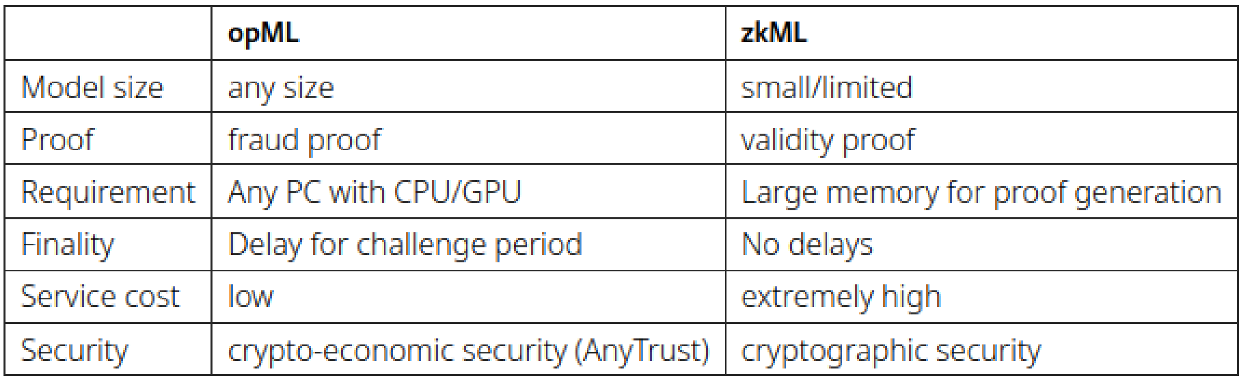
These are public parameters created by the setup of the ZK system.
The proving key is used by the prover to create a proof (for example that a certain inference computation was done correctly)
The verification key is used by the verifier to run the verification algorithm which checks the proof.

Optimistic Machine Learning

See Article
See Repo
See Paper

OPML enables off-chain AI model inference using an optimistic approach with an on chain interactive dispute engine implementing fault proofs.
This then relies on cryptoeconomics (and a blockchain) rather than a zk proof.





	opML	zkML
Model size	any size	small/limited
Proof	fraud proof	validity proof
Requirement	Any PC with CPU/GPU	Large memory for proof generation
Finality	Delay for challenge period	No delays
Service cost	low	extremely high
Security	crypto-economic security (AnyTrust)	cryptographic security

Timeline

A very new area driven by developments, particularly in ZKP.

2014 - Neural networks on encrypted data
2016 - zkSNARKS become feasible and are used with blockchain
2017 - Privacy preserving ML with MPC
2020 - vCNN - verifiable computation of CNNs using SNARKS
2020 - Circom libraries for ML
2022 - Prototype from Worldcoin using Plonky2

Challenges

Complexity

Although we are combining 2 complex areas, which are relatively new, when creating a proof we do not need to understand what the computation represents. We are purely interested in the fact that the computation performed has been done correctly.

Floating point / Integer representation

We often encode model parameters such as weights as 32 bit floating point numbers for precision, to be used with a zk circuit these need to be represented as integers in a finite field.
Quantisation or lower precision may be used to reduce the overhead needed for conversion.
Work is being done to improve this

- The cryptography underlying ZKPs is improving over time
- Projects such as taichoma include tools to provide more efficient conversion.
- Weightless approaches to NNs are being investigated, for example Zero Gravity

Non determinism

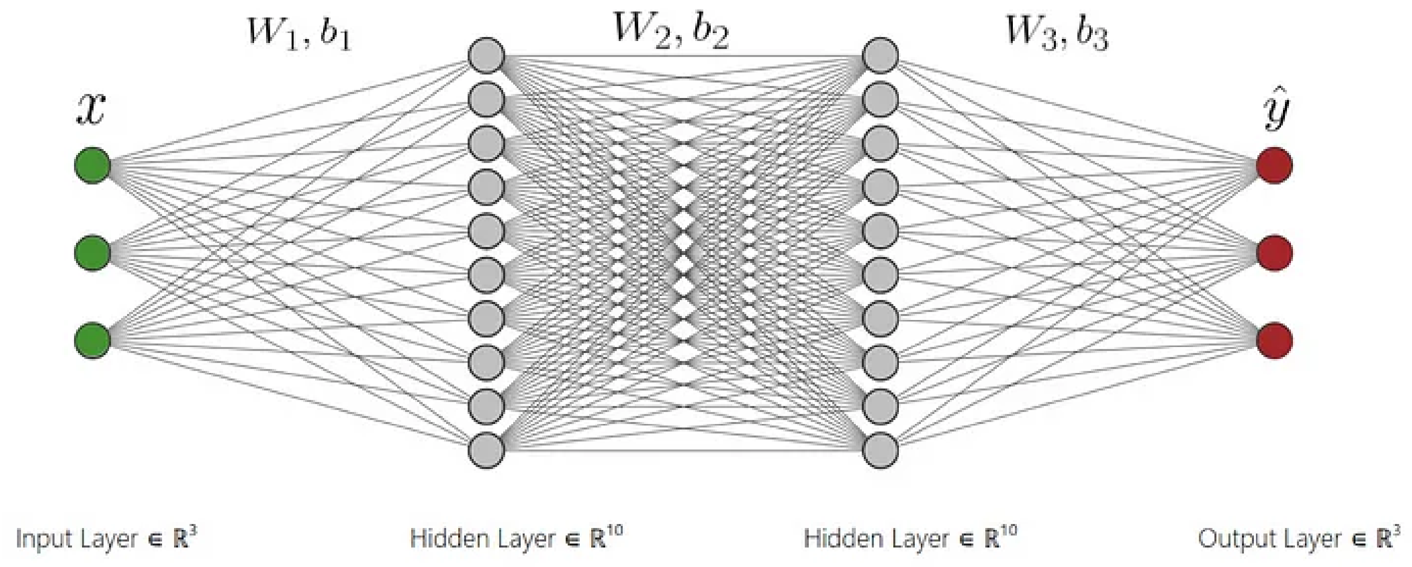
Related to floating point representation is the problem of determinism.
When verifying a computation we assume determinism, but ML models include randomness that is hard to account for, and floating point operations can produce different results on different hardware (and non deterministic results generally)

From article
"In short, decentralized inference networks are hard because all the details matter, and reality has a surprising amount of detail."

Representation of a NN as a circuit

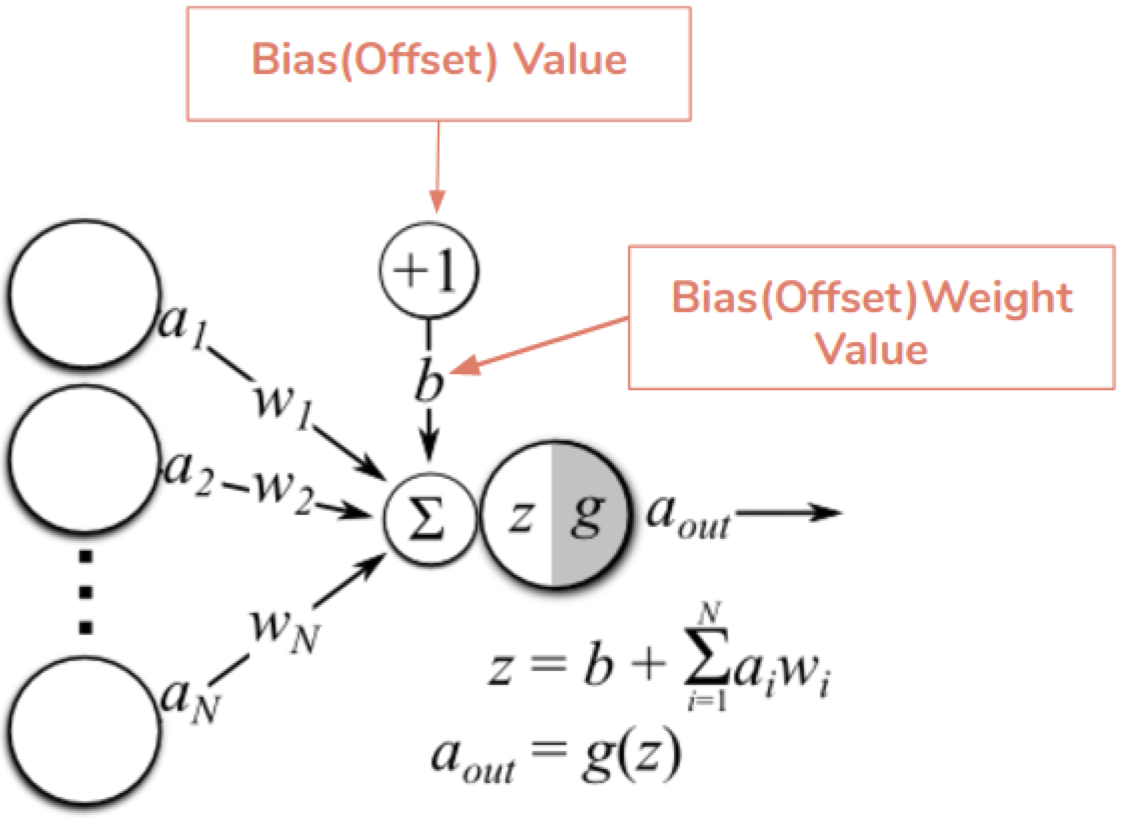
From a16zcrypto Blog

"When a model is represented as an arithmetic circuit, model parameters, constraints, and matrix multiplication operations may need to be approximated and simplified. And when arithmetic operations are encoded as elements in the proof's finite field, some precision might be lost (or the cost to generate a proof without these optimisation with current zero-knowledge frameworks would be too high)."


W_1, b_1
W_2, b_2
W_3, b_3
x
$\hat{y}$
Input Layer ∈ $\mathbb{R}^3$
Hidden Layer ∈ $\mathbb{R}^{10}$
Hidden Layer ∈ $\mathbb{R}^{10}$
Output Layer ∈ $\mathbb{R}^3$

Weights and Biases

See article

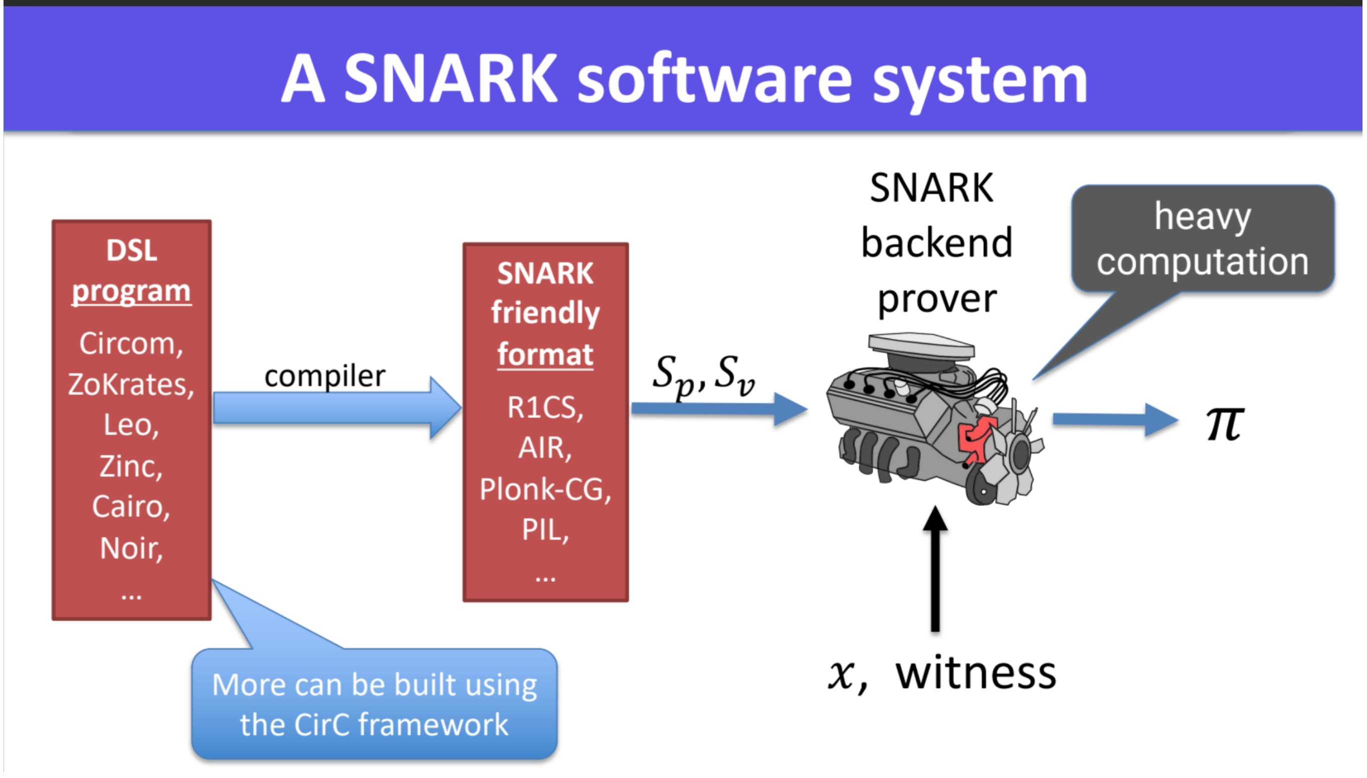


ZK Circuits

For zkSNARKS we construct arithmetic circuits to represent provable programs, in their simplest form these circuits consist of generic addition and multiplication gates.

Arithmetic circuit introduction

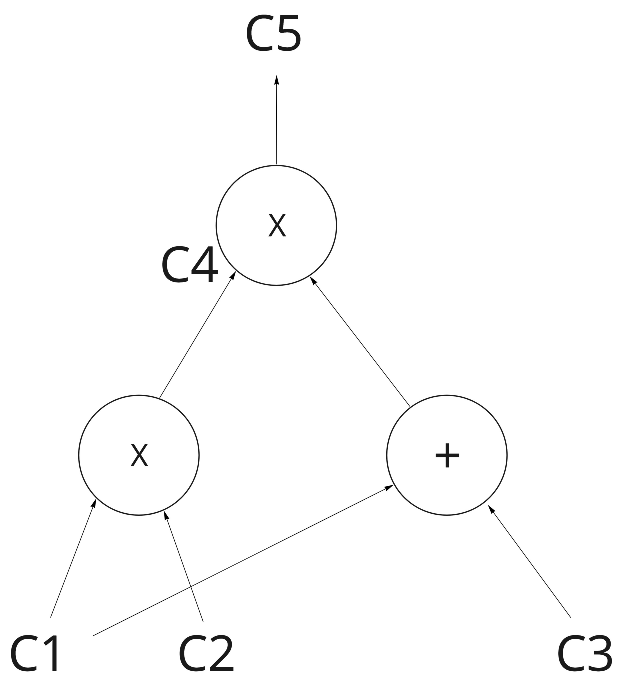
Slide from Stanford Web3 MOOC



An arithmetic circuit is one way to represent our DSL.
Arithmetic circuits are useful as they represented as a rank-1 constraint system, which can be transformed into polynomials, which is the next stage in the creation of the proof.

General form of an arithmetic circuit

We can visualise our circuit as a number of addition and multiplication gates.
These simple elements can be combined to represent any polynomial.



It is important to realise that this is not an algorithm, it is more like a hardware circuit. We can think of the circuit as having constraints, which will be satisfied if the correct inputs to the gates are supplied, i.e. we have the correct witness.

> When we implement a circuit we use large tables, where a gate is represented by a row in the table.
> When we are considering the interaction between the rows, this may be limited to nearby rows, but this varies with different aithmetisations.

Given the surface similarity these have to a neural network we may be tempted to represent a NN using generic gates, however this approach gives us performance problems.

See article by Pratyush Ranjan Tiwari

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 $w \times w$.
This is inefficient for any powerful CNN with more than a couple layers.

We need therefore to apply other techniques to reduce the complexity
Some attempts that have succeeded have been :

- Compute convolutions using Fast Fourier Transforms: See this paper which results in a circuit of size $O(n^2 \log n)$ with $O(\log n)$ depth.
- Use the sumcheck protocol, see this paper / recursion schemes / IVC
- Do some computation outside the circuit and check the result in the circuit
- Use lookup tables

Overview of lookup tables

SNARK unfriendly operations include operations with large overhead, are ones involving bit decomposition like bitwise XOR or AND.
For commonly used operations we precompute a lookup table of the legitimate (input, output) combinations; and the prover argues the witness values exist in this table.

Example

Let's consider a straightforward example. A prover and verifier want to determine whether an integer value is smaller than an arbitrarily chosen number, such as '32'. Accomplishing this using a constraint-only system is highly resource-intensive. The circuit would need to verify the binary representation of all input integers, ensuring that they do not exceed a specific bound. This results in numerous constraints and slower prover times.

Lookup tables can help us bypass this issue. A prover and verifier can begin with a pre-computed table of all 32 possible values. During the proof, they can simply ask if the value is in the table. If it is, then it must be an integer value under '32'. Lookup arguments have proven useful for a variety of non-ZKP-friendly operations, such as range checks and bitwise operations.

Lookup tables (or lookup arguments)

Overview of lookup tables

SNARK unfriendly operations include
Examples of operations with large overhead, are ones involving bit decomposition like bitwise XOR or AND.
For commonly used operations we precompute a lookup table of the legitimate (input, output) combinations; and the prover argues the witness values exist in this table.

Example
Let's consider a straightforward example. A prover and verifier want to determine whether an integer value is smaller than an arbitrarily chosen number, such as '32'. Accomplishing this using a constraint-only system is highly resource-intensive. The circuit would need to verify the binary representation of all input integers, ensuring that they do not exceed a specific bound. This results in numerous constraints and slower prover times.

Lookup tables can help us bypass this issue. A prover and verifier can begin with a pre-computed table of all 32 possible values. During the proof, they can simply ask if the value is in the table. If it is, then it must be an integer value under '32'. Lookup arguments have proven useful for a variety of non-ZKP-friendly operations, such as range checks and bitwise operations.