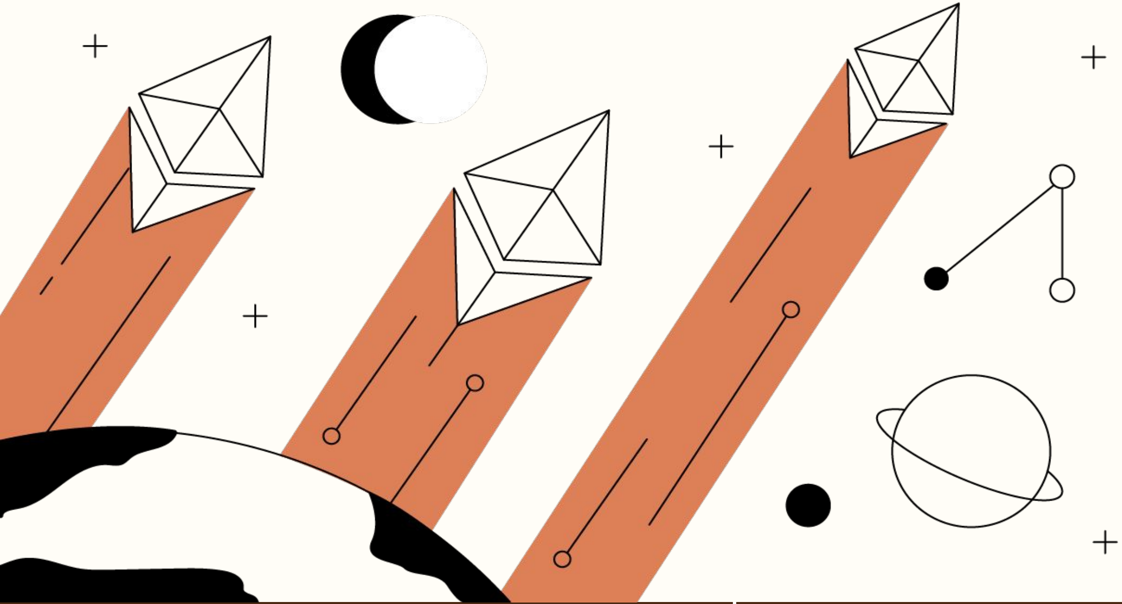
PLONK: A Universal zkSNARK Proof System

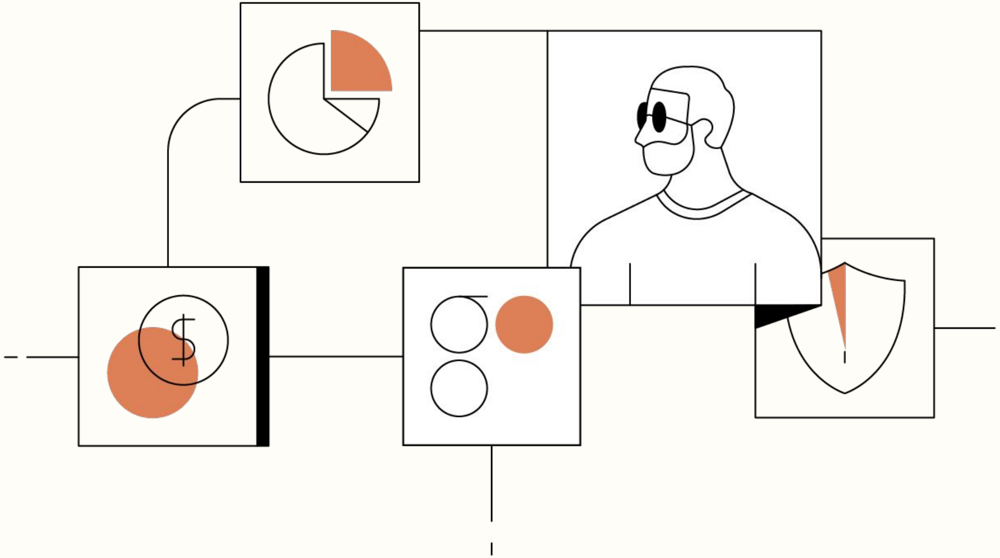
Hank Korth, Roberto Palmeri,
Tal Derei, dePaul Miller, Maxim Vezenov

Agenda



1. Primer into **Zero-Knowledge**
2. **General Background**: Proof Systems
3. **PLONK**
4. **Tutorial**: Circom and SnarkJS

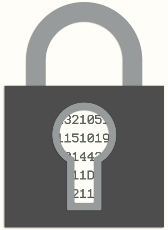
1. Zero-Knowledge



What is Zero Knowledge?

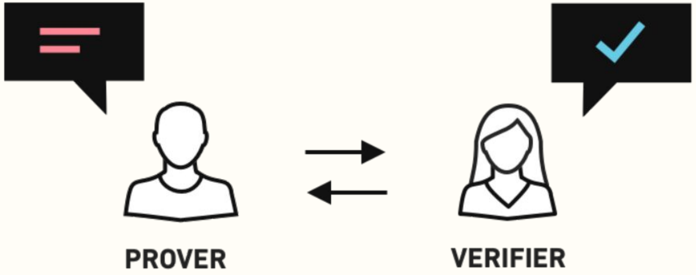
Zero Knowledge: "Way for a prover to convince verifier that something is true without revealing anything about why it's true."

Rooted in advanced mathematics and cryptography!



zk-SNARKs

- **zkSNARKs** = cryptographic **proofs**
  - Enables a prover to prove a mathematical statement to a verifier with a *short proof* and *succinct verification* using zero knowledge techniques.

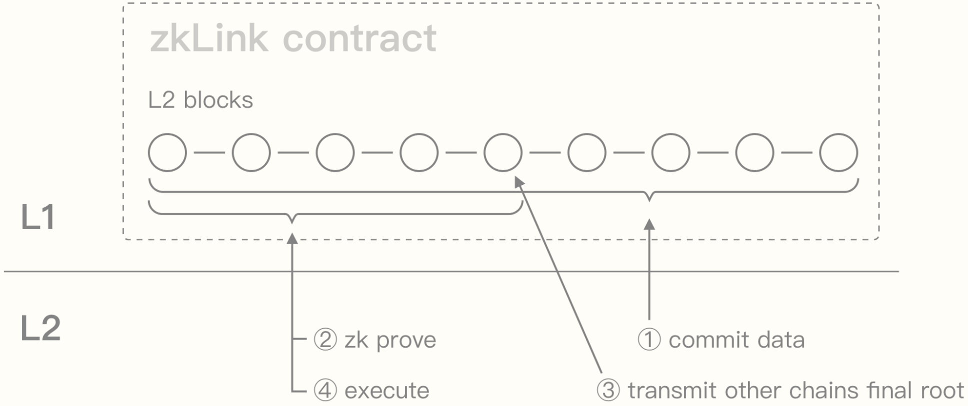


Non-Blockchain Example

Where's Waldo?

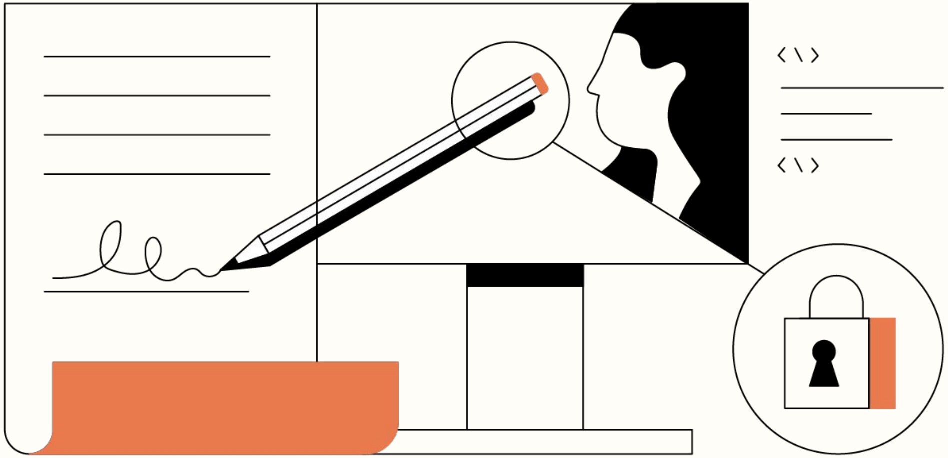
Blockchain Example

- **zk-Rollups** = Type of Layer-2 scaling solution that use zero-knowledge proofs to prove to the Layer-1 blockchain that transactions were executed correctly.



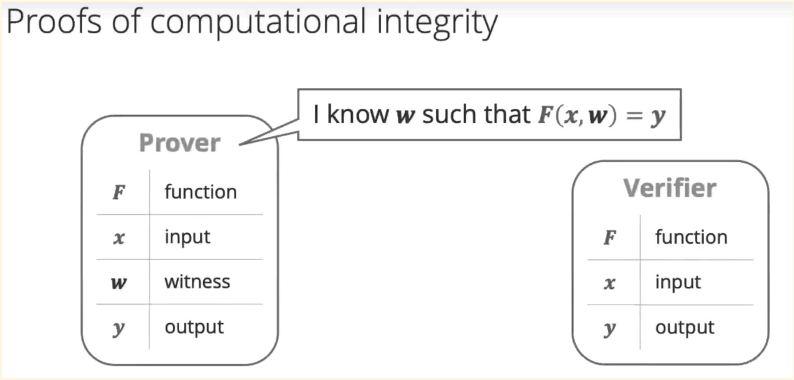
...ZKP used for for **verifiable computation** and **scalability**!

2. General Background: Proof Systems

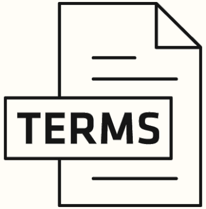


Generic Proof System Mechanics

Zero-Knowledge Proof System: A system capable of cryptographically generating a proof and verifying a computation, without revealing the solution.



Generic Proof System Mechanics



3 Steps!

- **Arithmetic Circuits**
- **Arithmetization**
- **Polynomial Commitments**

Arithmetic Circuits

<u>Arithmetic Circuit</u>: A program you want to generate a proof for.

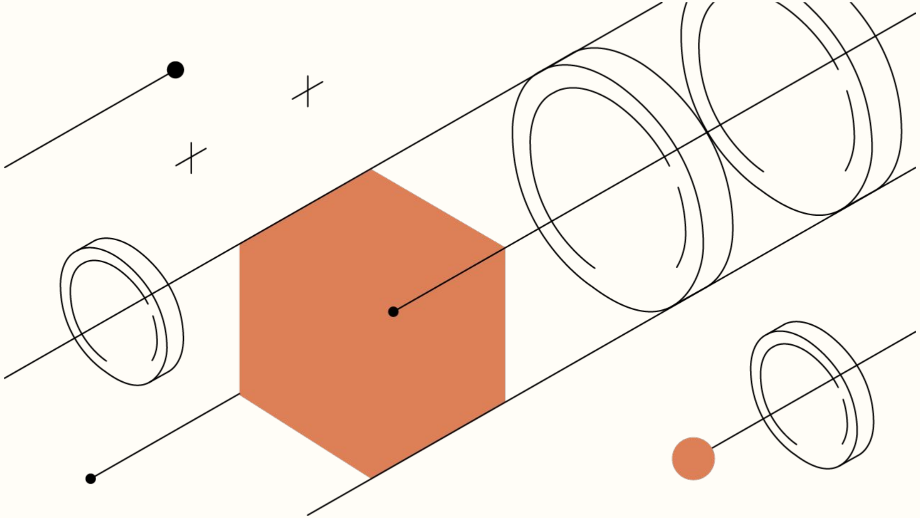
Circuits consist of **constraints** which must be of the form A*B + C = 0, where A, B and C are linear combinations of **signals (i.e. wires)**.

Circom Circuit

```
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;
}
```

Arithmetic Circuits

But how do we generate a <u>proof</u> for an arbitrary <u>circuit</u>?



Arithmetization

We need to first <u>Arithmetize</u> the circuit!

...

The circuit defines some statement you want to prove, and arithmetization <u>*transforms*</u> a statement so we can do math on it, for example:

[1] “I know some value ‘c’ that is the product of ‘a’ and ‘b’”

→

[2] “I know some polynomials that satisfies some polynomial identity”

Arithmetization

Arithmetization: Process of converting a circuit into an algebraic representation described by polynomials.

Arithmetization itself is composed of two steps:

[1] Generating the execution trace and polynomial constraints

[2] Transforming these two objects into a single low-degree polynomial

Arithmetization

How does this apply back to prover-verifier?

→ prover and the verifier agree on what the polynomial constraints are in advance.

→ prover then generates an execution trace of the program, and tries to convince the verifier that the polynomial constraints are satisfied over this execution trace, unseen by the verifier.

Example

Concrete Example: Supermarket Receipt

item	price
Avocado	$4.98
Apple	$7.98
Milk	$3.45
Bread	$2.65
Brown Sugar	$1.40
total	**$20.46**

Statement: prove the total sum we should pay at the market was computed correctly!

Arithmetization

- **Proof** = Receipt
- **Naive Verification**: Compute the total sum by going over every item in the list, and check it against the number at the bottom of the receipt
- **Succinct Verification**: Arithmetization (Execution Trace + Polynomial Constraints)

Arithmetization

<u>Execution Trace</u>: Table that represents all the steps of the underlying computation

item	price	running total
Avocado	\$4.98	\$0.00
Apple	\$7.98	\$4.98
Milk	\$3.45	\$12.96
Bread	\$2.65	\$16.41
Brown Sugar	\$1.40	\$19.06
total	**\$20.46**	\$20.46

Adding "running total" column allows us to verify each row individually, given its previous row.

Notice that the same constraint is applied to each pair of rows.

Arithmetization

<u>Polynomial Constraints</u>: Rephrase the execution trace as a set of linear polynomial constraints in Ai,j.

1) $A_{0,2} = 0$ // We start the running total from 0.
2) $\forall\ 1 \le i \le 5 : A_{i,2} - A_{i-1,2} - A_{i-1,1} = 0$ // Each row's running total is correct.
3) $A_{5,1} - A_{5,2} = 0$ // The last running total is the total sum.

Now we can transform these **constraints** into **polynomials**, and play a challenge game between prover and verifier on those polynomials.

This challenge game is done using **polynomial commitments**!

Just to Summarize Until Now

Just to summarize until this point…

- We took a problem of verifying the correctness of a receipt (i.e our **proof**)
- Transformed the receipt into a succinctly testable **execution trace**
- Created **polynomial constraints** from the execution trace
- Generate **polynomials** that satisfy those constraints
- Play a **challenge game** between prover and verifier using those polynomials

Polynomial Commitments

- **Polynomial Commitments:** Allows a prover to publish a value (*commitment*), while keeping the value hidden to the verifier (*hiding*).
  - Prover commits to a polynomial P (i.e. bind original message with a polynomial)

PC Schemes	KZG10	IPA	FRI	DARKS
Low level tech	Pairing group	Discrete log group	Hash function	Unknown order group
Setup	**G1, G2** groups **g1, g2** generators **e** pairing function s_k secret value in F	**G** elliptic curve g^n independent elements in G	**H** hash function w unity root	**N** unknown order **g** random in N **q** large integer
Commitment	$(a_0s^0 + \ldots + a_ns^n)g_1$	$a_0g_0 + \ldots + a_ng_n$	$H(f(w^0), \ldots, f(w^n))$	$(a_0q^0 + \ldots + a_dq^d)g$

Different ZKPs have different Commitment Schemes

Polynomial Commitment

→ Prover commits to certain polynomial **P** (bind original message to polynomial)

→ Prover proves the value of polynomial at certain point **Z** satisfies **P(Z)** through the proof <u>WITHOUT</u> revealing the polynomial

$$\boxed{\mathbf{P(Z) = z}}$$

Polynomial Commitment

A polynomial commitment is a sort of "<u>hash</u>" of some polynomial P(x) with the property that you can perform arithmetic checks on hashes.

```
Polynomial Commitments:
h_P = commit(P(x)) on P(x)
h_Q = commit(Q(x)) on Q(x)
h_R = commit(R(x)) on R(x)
```

<u>**We can show**</u>:

- If **P(x) + Q(x) = R(x)** *OR* **P(x) * Q(x) = R(x)**, you can generate a proof that proves this relation against h_P, h_Q, h_R
- If P(z) = a you can generate a proof (known as an "opening proof") that the evaluation of **P** at **z** is indeed **a**

Polynomial Relationships

…What's the point?

There exists verifiable relationships between **polynomials**!

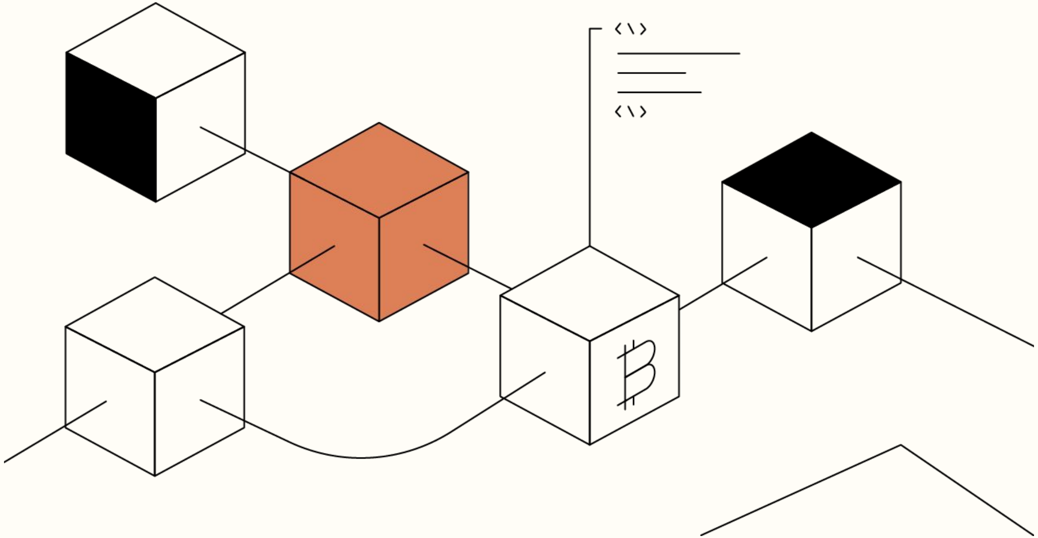
.

.

.

Therefore a prover can convince a verifier using **PLONK proofs** composed of these polynomials!

3. PLONK



The Roadmap

→ PLONK **Overview:** Features and Optimizations

→ PLONK **Math**: A Recipe and Ingredients

→ Final **Intuition** about PLONK

PLONK Overview

Trusted Setup

To understand **PLONK**, you have to first understand the concept of a **Trusted Setup**

…and how PLONK <u>improves</u> it.

Trusted Setup

Trusted Setup: a complex coordinated event that generates the public parameters that kick off SNARK-based systems. These events are known as **Multi-Party Computation (MPC) Ceremonies**.

.
.
.

These public parameters are the called **reference strings**, used to construct the **private keys** used to generate and verify proofs for circuits!

PLONK is **universal**, so it only has to do this setup once for all circuits!

Three Flavours of SNARKs

Non-Universal

→ Example: **Groth16**

→ **Hard** to coordinate

→ **Medium** storage costs

→ **Hard** to update code

Universal

→ Example: **PLONK**

→ **Easy** to coordinate

→ **Medium** storage costs

→ **Easy** to update code

Transparent

→ Example: **STARKs**

→ **Easy** to coordinate

→ **Low** storage costs

→ **Easy** to update code

PLONK Proof System

PLONK: A Universal SNARK Proof System
[Developed by **AZTEC** and **Protocol Labs** in **2019**]

IOP Layer: PLONK Core

Accumulation Layer: Recursive Proofs

Polynomial Commitment Layer: KZG

PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge

Ariel Gabizon* — Aztec
Zachary J. Williamson — Aztec
Oana Ciobotaru

April 27, 2022

Protocol Labs aztec

PLONK Features and Optimizations

- **Universal Setup** - updatable reference string, and concise proofs (~1kb)
- PLONK circuits described/structured as a collection of **gates**:
  multiplication (×) and *addition* (+).

- **<u>Optimizations</u>:**
  - **TurboPLONK** extends this by introducing **custom gates**: MIMC hash elliptic curves operations, etc.
  - **UltraPLONK** then adds support for **plookup** (i.e. lookup tables).
  - **SHPLONK** optimizes on polynomial commitment layer by achieving smaller proof sizes and shorter proving times.
  - Another optimization is **recursive proof composition**, which allows you to aggregate multiple proofs and compress it into a single proof.

Protocol Labs aztec

PLONK Attributes

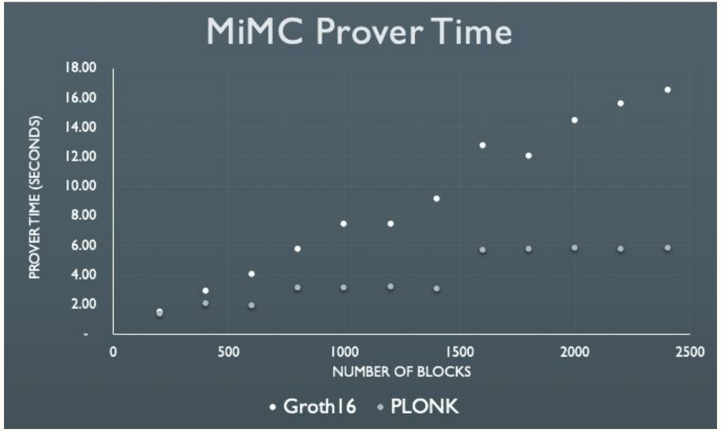
<u>Proof Generation:</u>	Log-Linear *O(n log n)* for <u>ALL</u> zk-SNARKs
<u>Proof Verification:</u>	Logarithmic O(log n)
<u>Proof Size:</u>	0.5-1 KB
<u>Trusted Setup:</u>	**Yes**, <u>Universal</u>

Protocol Labs aztec

PLONK Benchmarks I

TurboPLONK **2.5x faster** than Groth16 on MiMC Hashes

	PLONK	Groth16
MiMC Prover Time	5.6s	16.5s
SHA-256 Prover Time	6.6s	1.4s
Verifier Gas Cost	223k	203k
Proof Size	0.51kb	0.13kB

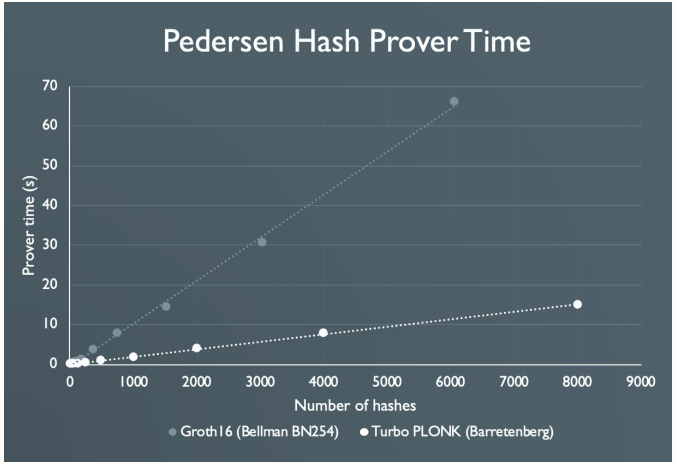


Protocol Labs aztec

PLONK Benchmarks I

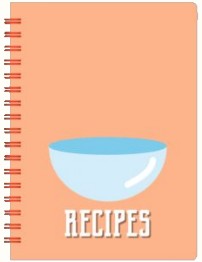
TurboPLONK **~5x faster t**han Groth16 on Pedersen Hashes

	PLONK	Groth16
Pedersen Prover Time	4.7s	23.9s
MiMC Prover Time	5.8s	16.5s
SHA-256 Prover Time	2.1s	1.4s
Verifier Gas Cost	223k	203k
Proof Size	0.51kB	0.13kB



Protocol Labs aztec

PLONK Math: A Recipe



PLONK Recipe

We need to **transform** an arbitrary program into something a proving system can understand…so we can generate a **proof**!

…

These ZKP constructions understand **polynomials** under the hood.

PLONK Recipe

How do we generate a PLONK proof for an arbitrary program? What are the ingredients needed?

1. Circuit: Represent your program as an arithmetic circuit (i.e. gates).

2. Arithmetization: Convert your circuit description into a polynomial identity / relationship.

3. Polynomial Commitments: Evaluate the polynomial identity using a succinct polynomial commitment scheme.

The Final Product

PLONK is fundamentally a **protocol** to prove:

$$f(x) = 0, \forall\, x \in H,\, H = \{h_1, h_2, \ldots, h_n\}$$

Prover sends that polynomial to verifier, who verifies:

$$f(h_1) = 0$$
$$f(h_2) = 0$$
$$.$$

we can find the number of polynomial roots by,

$$.$$
$$f(h_n) = 0$$

But there's a more ***succinct*** verification method:

$$f(x) = (h_1 - x)(h_2 - x)\ldots t(x)$$

Where the product $Z_H(x)$ is **vanishing** polynomial of domain H, and $t(x)$ is the **quotient polynomial.**

So if we have $f(x)$ and we divide it by the vanishing polynomial $Z_H(x)$, the remainder is the quotient polynomial $t(x)$.

Now comes in the **Schwartz-Zippel Lemma:**

$$\textit{polynomial}\ f = g,\ \textit{then}\ f(x) = g(x)\ \forall\ x$$

And

$$L(x) = f(x) - g(x) = 0$$

Using this knowledge, we can show: **[*SEE NEXT SLIDE*]**

The Final Product

ZKP Protocol: Trying to prove that polynomial is vanishing in the domain of H

PROVER **VERIFIER**

$r \in \mathbb{Z}$ ←

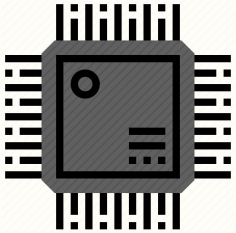
$\mathit{evaluations\ at}\ f(r),\ t(r)$ →

Verifier checks whether $f(r) = Z_H(r) \cdot t(r)$

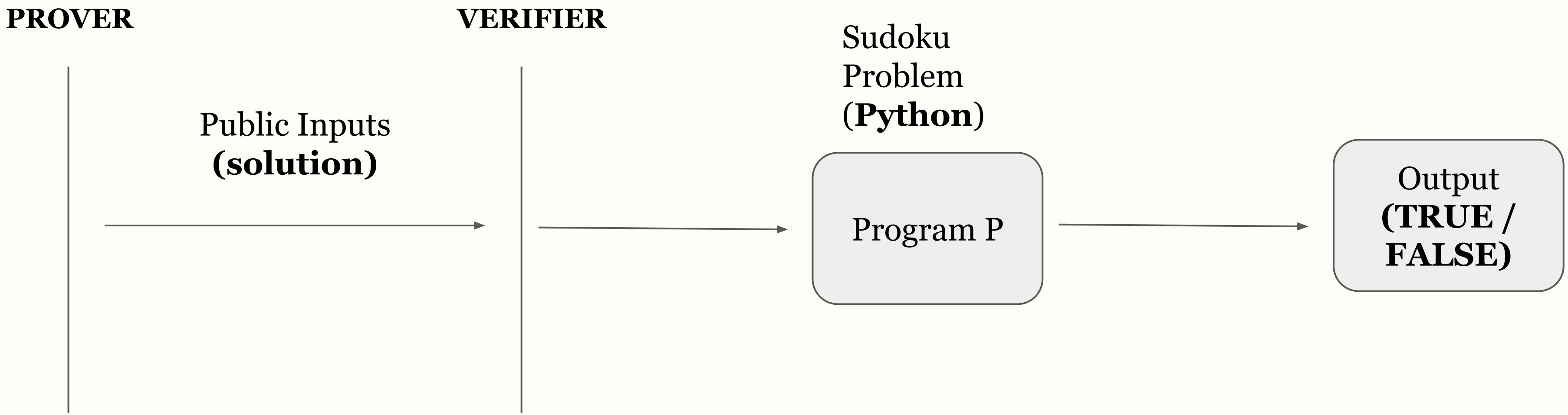
Instead of checking all the evaluations of domain H, we're checking the evaluation of two polynomials at random points.

This is **succinct verification**!

Programs → Circuits



PLONK Circuits



...Now we convert the program into an **arithmetic circuit** composed of **gates**!

PLONK Circuits

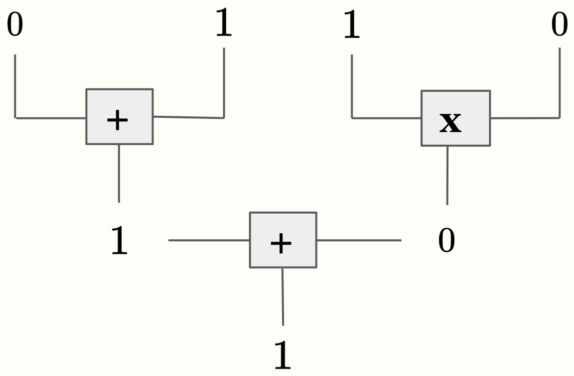
PROVER

VERIFIER

Private inputs
(**solution**)

Program P
(Arithmetic Circuit)

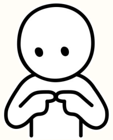
Output
(TRUE / FALSE)



PLONK Circuits

- Arithmetic Circuits described with + and * **gates**

- **Addition** gates aren't free 

- ...but **'custom'** gates are! 

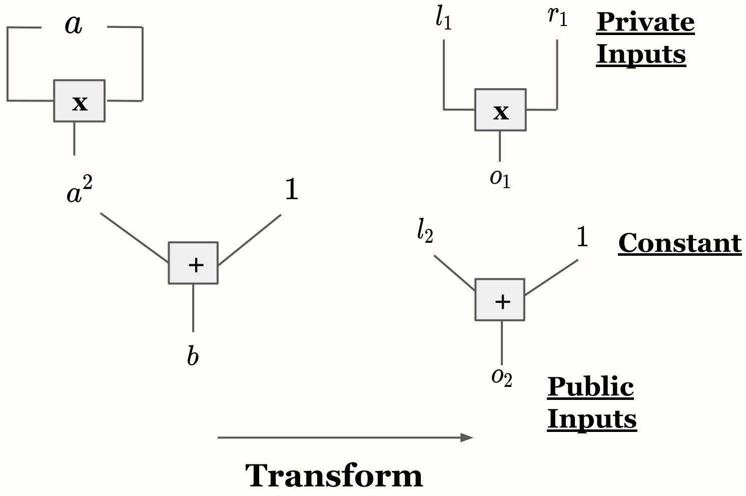
Arithmetization:

Arithmetic Circuits —> Constraint System

PLONK Constraint System

Prover: knows a such that $b - 1 = a^2$. For example, **a = 5, b = 26.**

Arithmetic Circuit:



Let's write these constraints as set of equations, known as a **constraint system**:

$$(1)\ l_1 \cdot r_1 - o_1 = 0$$
$$(2)\ l_2 + 1 - o_2 = 0$$

and set of **copy constraints**:

$$l_1 = r_1$$
$$o_1 = l_2$$

Now we normalize these equations before we can convert them to polynomials. PLONK has a special equation to do it:

$$l_i \cdot q_{L_i} + r_i \cdot q_{Ri} + o_i \cdot q_{o_i} + q_{c_i} + l_i \cdot r_i \cdot q_{M_i} = 0$$

$$5 \cdot 0 + 5 \cdot 0 - 25 \cdot 1 + 0 + 5 \cdot 5 \cdot 1 = 0$$

$$-25 \cdot 1 + 0 \cdot 0 - 26 \cdot 1 - 1 - 25 \cdot 0 \cdot 0 = 0$$

Constraint System → Polynomials

PLONK Circuits

View as a **table**, and treat all the columns as separate **vectors:**

$l \cdot q_L$	$+ r \cdot q_R$	$+ o \cdot q_o$	$+ q_c$	$+ l \cdot r \cdot q_M$	$= 0$
$5 \cdot 0$	$+5 \cdot 0$	$- 25 \cdot 1$	$+0$	$+5 \cdot 5 \cdot 1$	$= 0$
$-25 \cdot 1$	$+0 \cdot 0$	$-26 \cdot 1$	-1	$-25 \cdot 0 \cdot 0$	$= 0$

So the **vectors** look like this:

$$\begin{aligned} l &= (5, -25) \\ q_L &= (0, 1) \\ &\dots \end{aligned}$$

Now we convert the vectors into polynomials, known as **interpolation.**

So let's create a domain $H = \{h_1, h_2\}$ in a field F.

Now let's take a polynomial $l(x)$ such that

$$\begin{aligned} l(1) &= 5 \\ l(2) &= -25 \end{aligned}$$

So we're **interpolating** the vector l into $l(x)$ lagrange polynomials:

Now, let's do it for all vectors.

$$\begin{aligned} f(x) = \; & l(x) \cdot q_L(x) + r(x) \cdot q_R(x) + o(x) \cdot q_0(x) \\ & + q_c(x) + l(x) \cdot r(x) \cdot q_M(x) = 0 \end{aligned}$$

and we compress circuit into single polynomial!

PLONK Circuits

So where are we?

→ We have **compressed** an entire circuit into a single polynomial!

and

→ The verifier needs to **verify** that the prover's polynomial, which represents the execution of the circuit, is equal to 0!

→ Let's see what the PLONK protocol does with this polynomial.

The Protocol

The Protocol

PLONK is fundamentally a **protocol** to prove:

$$f(x) = 0, \forall x \in H, H = \{1, 2\} \in F$$

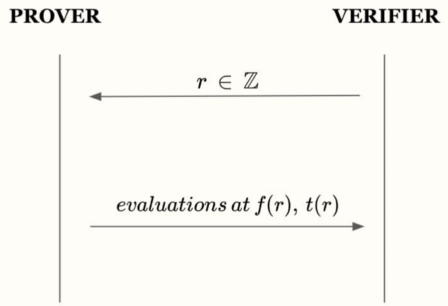
We can rewrite the polynomial as:

$$f(x) = (1 - x)(2 - x)\ldots t(x)$$

Where the product $Z_H(x)$ is **vanishing** polynomial of domain H, and $t(x)$ is the **quotient polynomial.**

...

ZKP Protocol: Trying to prove that polynomial is vanishing in the domain of H



Verifier checks whether $f(r) = Z_H(r) \cdot t(r)$

Instead of checking all the evaluations of domain H,
we're checking the evaluation of two polynomials at random points

This is **succinct verification**!

The Protocol

BUT…how do we know the prover sending f(r) and t(r) to the verifier is correct?

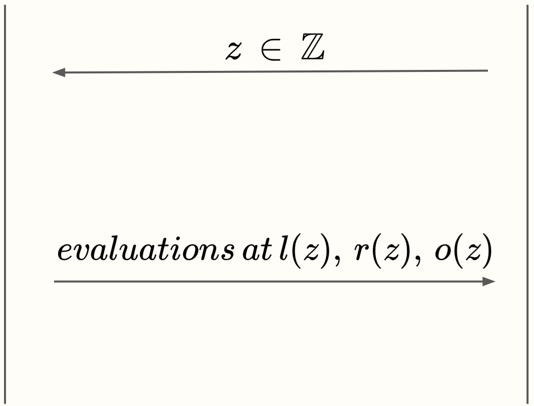
AND…The prover and verifier want to perform this polynomial dance in such a way that allows the prover to hides some parts of the polynomial.

So how do we perform this dance?

Polynomial Dance

Polynomial

$$f(x) = l(x) \cdot q_L(x) + r(x) \cdot q_R(x) + o(x) \cdot q_0(x) + q_c(x) + l(x) \cdot r(x) \cdot q_M(x) = 0$$



Verifier checks whether: $f(z) = Z_H(z) \cdot t(z)$

where $Z_H(z) \cdot t(z) = l(z) \cdot q_R(z) + r(z) \cdot q_R(z) + o(z) \cdot q_0(z) + q_c(z) + l(z) \cdot r(z) \cdot q_M(z)$

Polynomial Commitment Scheme (PCS)

Polynomial Commitments

Let's take the polynomial $l = l_0 + l_1 x + l_2 x^2 \ldots$

Then prover can **commit** the polynomial and publish the commitment

$$L = commit(l)$$

Verifier can then ask to **evaluate** the polynomial at some point z, and prover sends back:

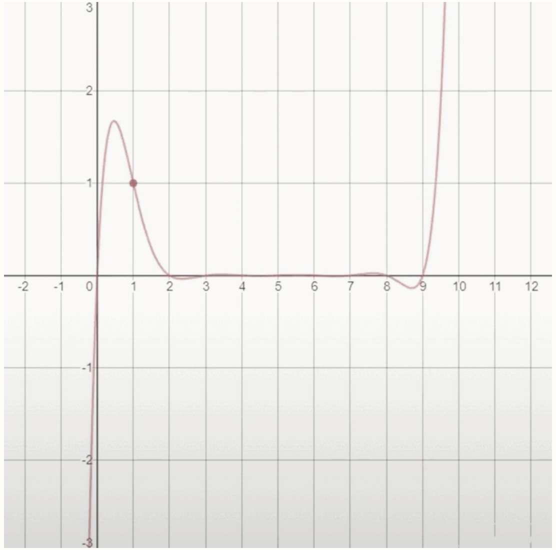
(1) The polynomial evaluated at z $l(z)$
(2) Proof π

PLONK uses the **KZG** **PCS**, requires **trusted setup**

Final Intuition

Lagrange Bases and Multiplicative Subgroups

Lagrange Bases: a different way of encoding a polynomial



4. **Tutorial:** Circom and SnarkJS

Tutorial

Tutorial on building circuits and generating **PLONK proofs** using *Circom* and *SnarkJS*:

- **Circom:** compiler written in Rust for compiling circuits written in the circom programming language. The compiler outputs the representation of the arithmetic circuit as a set of constraints.

- **SnarkJS**: a javascript and pure web assembly implementation of the Groth16/PLONK schemes, generating and validating proofs for circom circuits.

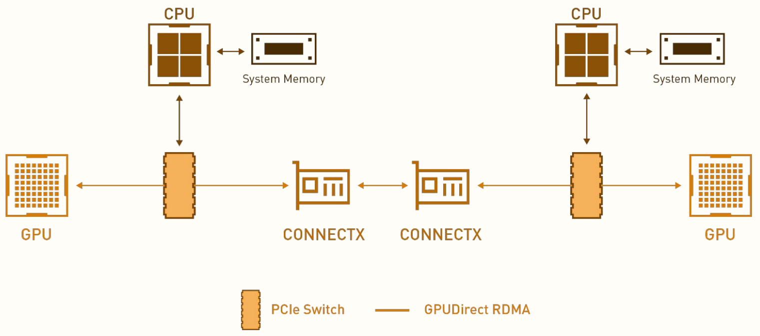
- **Circomlib**: a library of circom templates that contains hundreds of circuits such as comparators, hash functions, digital signatures, and many more.

Tutorial Link: https://github.com/TalDerei/PLONK-Tutorial

Future Research

Future research involves applying **RDMA** integration to PLONK and creating a **GPU-based PLONK prover**.

It would be interesting to see how this implementation scales for general computations, like arbitrary smart contract calls, on **Layer-2 zkEVMs**.



References

https://eprint.iacr.org/2019/953.pdf

https://www.youtube.com/watch?v=ty-LZf0YCK0

https://www.youtube.com/watch?v=n6_nicl4ckM

https://medium.com/starkware/arithmetization-i-15c046390862

https://aztec.network/research/

https://www.youtube.com/watch?v=bz16BURH_u8

https://www.youtube.com/watch?v=RUZcam_jrz0

https://hackmd.io/@aztec-network/plonk-arithmetiization-air

https://zeroknowledge.fm/news-2-on-optimization-plonk/

https://medium.com/aztec-protocol/plonk-benchmarks-2-5x-faster-than-groth16-on-mimc-9e1009f96dfe

https://medium.com/aztec-protocol/plonk-benchmarks-ii-5x-faster-than-groth16-on-pedersen-hashes-ea5285353db0

Thank you!

SCALABLE SYSTEMS
& SOFTWARE
RESEARCH GROUP

https://sss.cse.lehigh.edu/