Lesson 13

Week 4

Lesson 13 - What are ZK EVMs part 3 - proving system

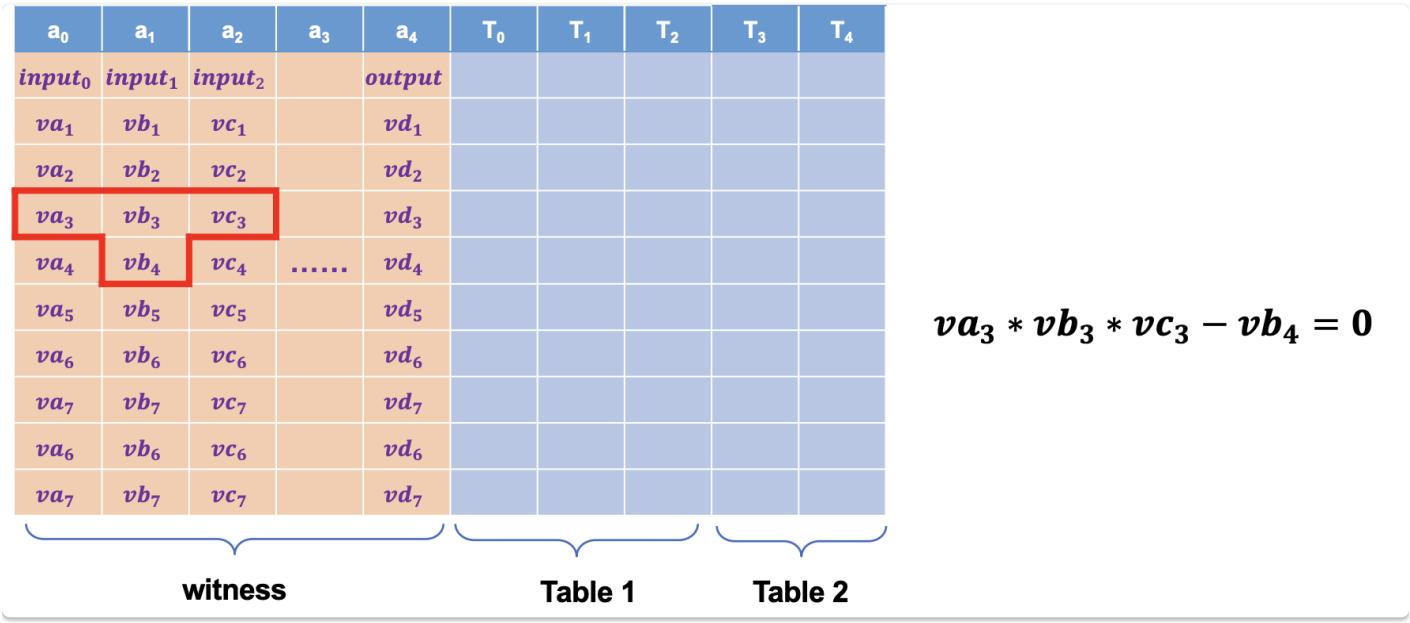
Lesson 14 - zkEVM security

Lesson 15 - Overview of Proving Systems

Lesson 16 - SNARK implementation

Scroll custom gate

Taken from presentation



$$va_3 * vb_3 * vc_3 - vb_4 = 0$$

Plonkish Arithmetization – Permutation

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	.....	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness Table 1 Table 2

$$vb_4 = vc_6 = vb_6 = va_6$$

Plonkish Arithmetization – Lookup argument

Scroll

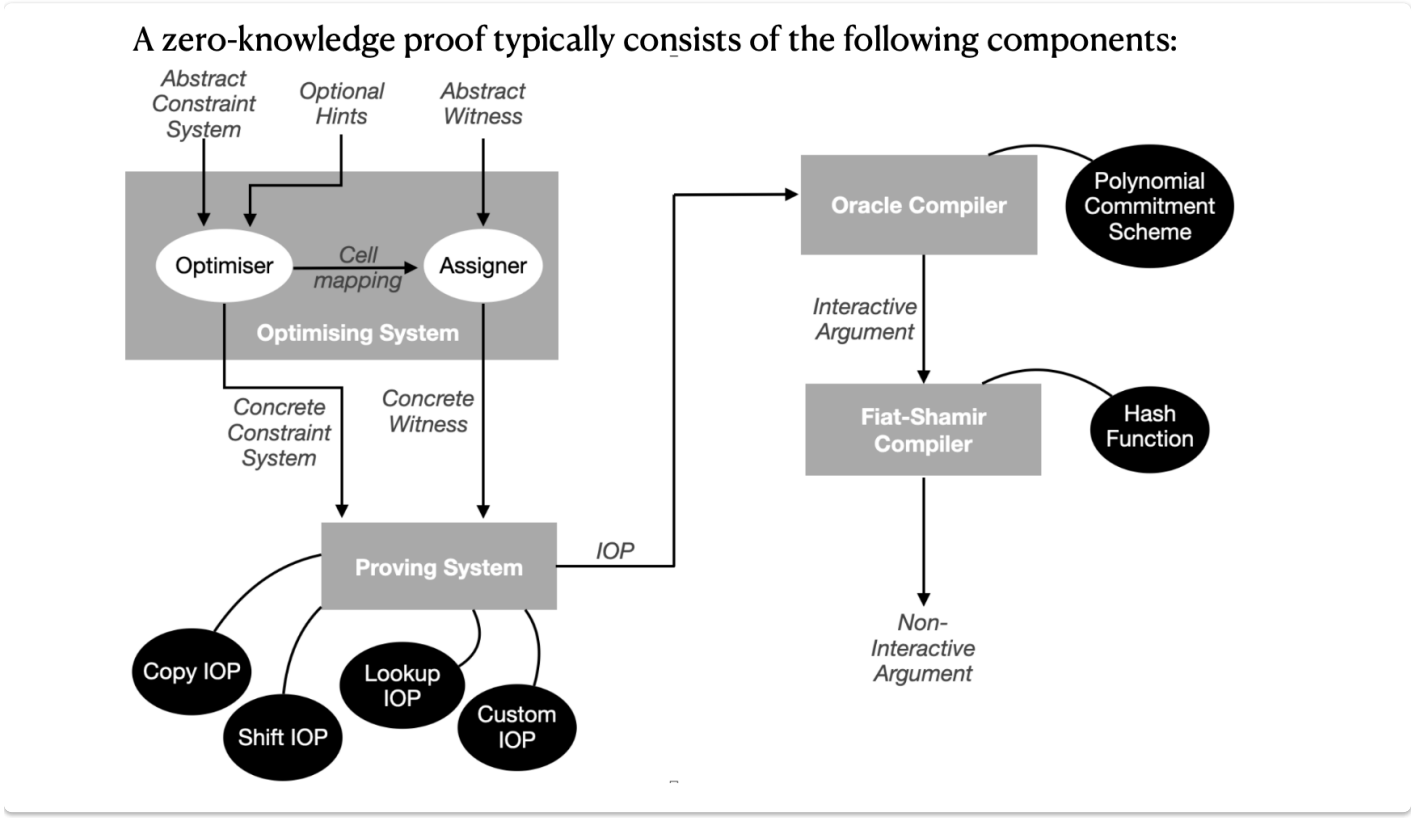
a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$	0000				
va_1	vb_1	vc_1		vd_1	0001				
va_2	vb_2	vc_2		vd_2	0010				
va_3	vb_3	vc_3		vd_3	0011				
va_4	vb_4	vc_4	.....	vd_4	0100				
va_5	vb_5	vc_5		vd_5	0101				
va_6	vb_6	vc_6		vd_6	.....				
va_7	vb_7	vc_7	Lookup		1101				
va_6	vb_6	vc_6		vd_6	1110				
va_7	vb_7	vc_7		vd_7	1111				

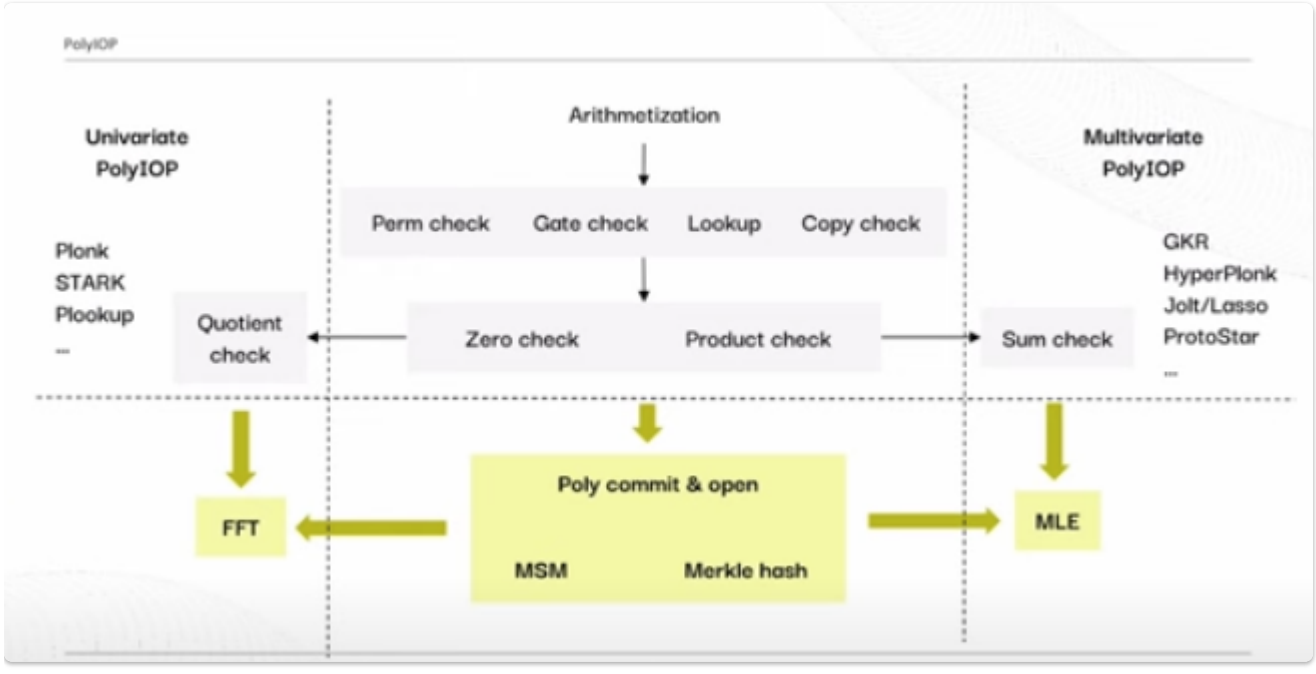
witness Table 1 Table 2

$$vc_7 \in [0, 15]$$

Process overview

Proving systems take different approaches, though there is some modularity involved.
The following 2 diagrams should give you a general idea of the process.





> A large part of this process for most recent proving systems involves Polynomial Commitment Schemes.

Polynomial Commitment Schemes

Introduction

A polynomial commiment is a short object that "represents" a polynomial, and allows you to verify evaluations of that polynomial, without needing to actually contain all of the data in the polynomial.

That is, if someone gives you a commitment c representing $P(x)$, they can give you a proof that can convince you, for some specific z, what the value of $P(z)$ is.
There is a further mathematical result that says that, over a sufficiently big field, if certain kinds of equations (chosen before z is known) about polynomials evaluated at a random z are true, those same equations are true about the whole polynomial as well.

For example, if $P(z).Q(z) + R(z) = S(z) + 5$ for a particular z, then we know that it's overwhelmingly likely that $P(x).Q(x) + R(x) = S(x) + 5$ in general.
Using such polynomial commitments, we could very easily check all of the above polynomial equations above - make the commitments, use them as input to generate z, prove what the evaluations are of each polynomial at z, and then run the equations with these evaluations instead of the original polynomials.

A general approach is to have the evaluations in a merkle tree, the leaves of which the verifier can select at random, along with merkle proof of their membership.

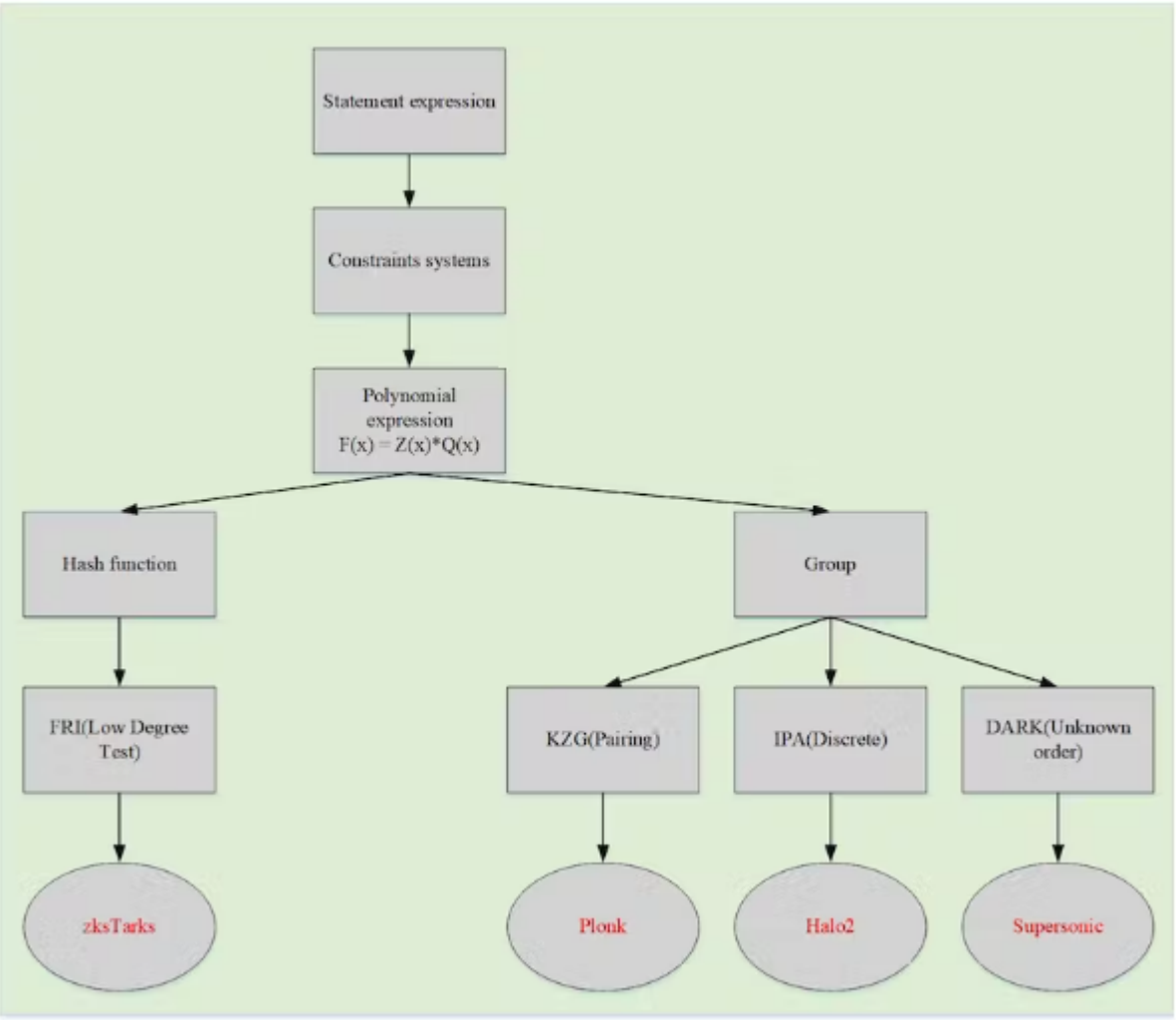
Role in ZKPs

Commitment schemes generally allow the properties of

1. Binding. Given a commitment c, it is hard to compute a different pair of message and randomness whose commitment is c.
   This property guarantees that there is no ambiguity in the commitment scheme, and thus after c is published it is hard to open it to a different value.
2. Hiding. It is hard to compute any information about m given c.

Given the size of the polynomials used in ZKPs , with say 10^8 terms, they help with succinctness by reducing the size of the information that needs to be passed between the prover and verifier.

Types of PCS



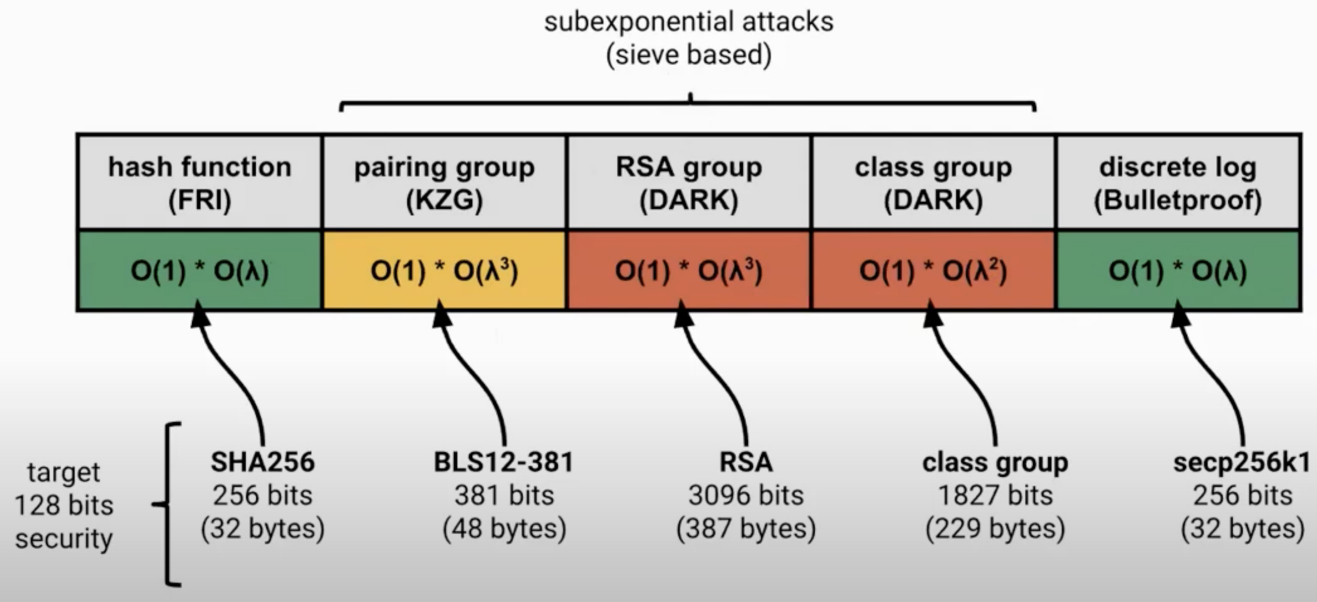
Comparison of Schemes and their underlying assumptions

Taken from [ZKP Study Group VideoSlides]

	hash function	pairing group	RSA group	class group	discrete log
transparent					
succinct					
unbounded					
updatable		curve-specific			
post-quantum					

	hash function	pairing group	UO group	discrete log group
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d*\log(d))$	$O(d)$	$O(d*\log(d))$	$O(d)$

	hash function (FRI)	pairing group (KZG)	UO group (DARK)	discrete log group (Bulletproof)
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d*\log(d))$	$O(d)$	$O(d)$	$O(d)$



KZG

See article

You could create a commitment by taking the coefficients of a polynomial and using these as the leaves of a merkle tree and giving the root as the commitment.
The problem with this simple approach is the amount of computation that needs to be done to show the proof, and the proof has the coefficients in the clear.

A better approach has the following features

1. The commitment size is one group element of an elliptic group that admits pairings. For example, with BLS12_381, that would be 48 bytes.
2. The proof size, *independent* from the size of the polynomial, is also always only one group element. Verification, also independent from the size of the polynomial, requires a two group multiplications and two pairings, no matter what the degree of the polynomial is.
3. The scheme *mostly* hides the polynomial – indeed, an infinite number of polynomials will have exactly the same Kate commitment. However, it is not perfectly hiding: If you can guess the polynomial (for example because it is very simple or in a small set of possible

polynomials) you can find out which polynomial was committed to.

Trusted setup procedure

1. From a field $\mathbb{F}_p$ we get a random point s
2. We then create $\{G, sG, s^2G, \dots\}$ up to the degree of the polynomials we are expecting.
   This is our structured reference string
3. The s value is the 'toxic waste' and is deleted
4. The prover will need to evaluate $f(s)$ later but doesn't know s, but does have the s values 'hidden' in the SRS

Homomorphic properties of commitments

We rely upon the fact that
$com(A) = com(B)$ iff $A = B$
and $com(A + B)$ = $com(A) + com(B)$

Creating the commitment

So our original polynomial is $f(x)$ = $f_0 + f_1x + f_2x^2 + \dots$
and the prover wants to evaluate this at s
We can rewrite our SRS as

$\{G, sG, s^2G, \dots\}$ = $\{T_0, T_1, T_2, \dots\}$

then we want to rewrite $f(s)$ in terms of
$f_0T_0 + f_1T_1 + f_2T_2 + \dots$
= $f_0G + f_1sG + f_2s^2G$
$= (f_0 + f_1s + f_2s^2 + \dots)G$
$= f(s)G$

this is our commitment to f
$= com(f)$
so our commitment is hidden behind the group element G and $f(s)G$ is a curve point.

Evaluation Proof

The verifier will want the commitment $com(f)$ and also $f(z)$
for some random $z \in \mathbb{F}_p$

Now the verifier wants an evaluation proof $f(z)$ given z and $com(f)$

The equation $f(x) - f(z)$ has a root at z
and we can do our rewrite technique to express this as $(x - z)h(x)$

So the verifier can ask for an evaluation of $f(z)$ knowing that $h(x)$ must exist if the equalities are all correct

IOP Introduction

With an IOP the verifier only gets oracle access to the prover message rather than full access that previous methods used.

Building proving systems with IOPs

See Notes from Berkley zkp_course

Choices

1 and 2 are *transparent*

1. Any polynomial IOP + IPA/Bulletproofs polynomial commitment.
   For example Halo2-ZCash
   Pros: Shortest proofs among transparent SNARKs.
   Cons: Slow
2. Any polynomial IOP + FRI PCS
   For example STARKs, Fractal, Aurora, Virgo, Ligero++
   Pros : Pros: Shortest proofs amongst plausibly post-quantum SNARKs
   Cons : Cons: Proofs are large (100s of KBs depending on security)
3. Linear-PCP based
   For example Groth16
   Pros : Pros: Shortest proofs (3 group elements), fastest V

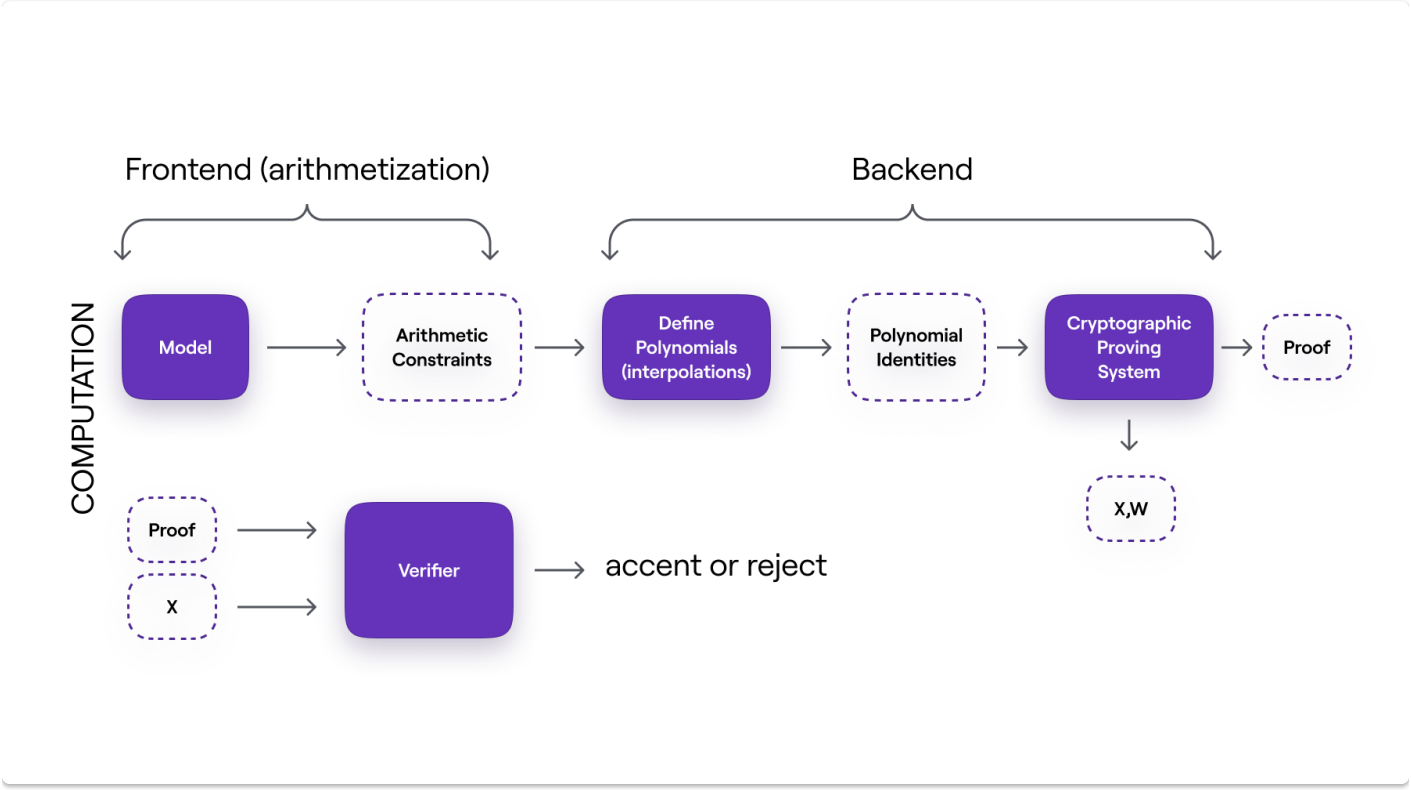
Cons: Circuit-specific trusted setup, slow and space-intensive P, not postquantum

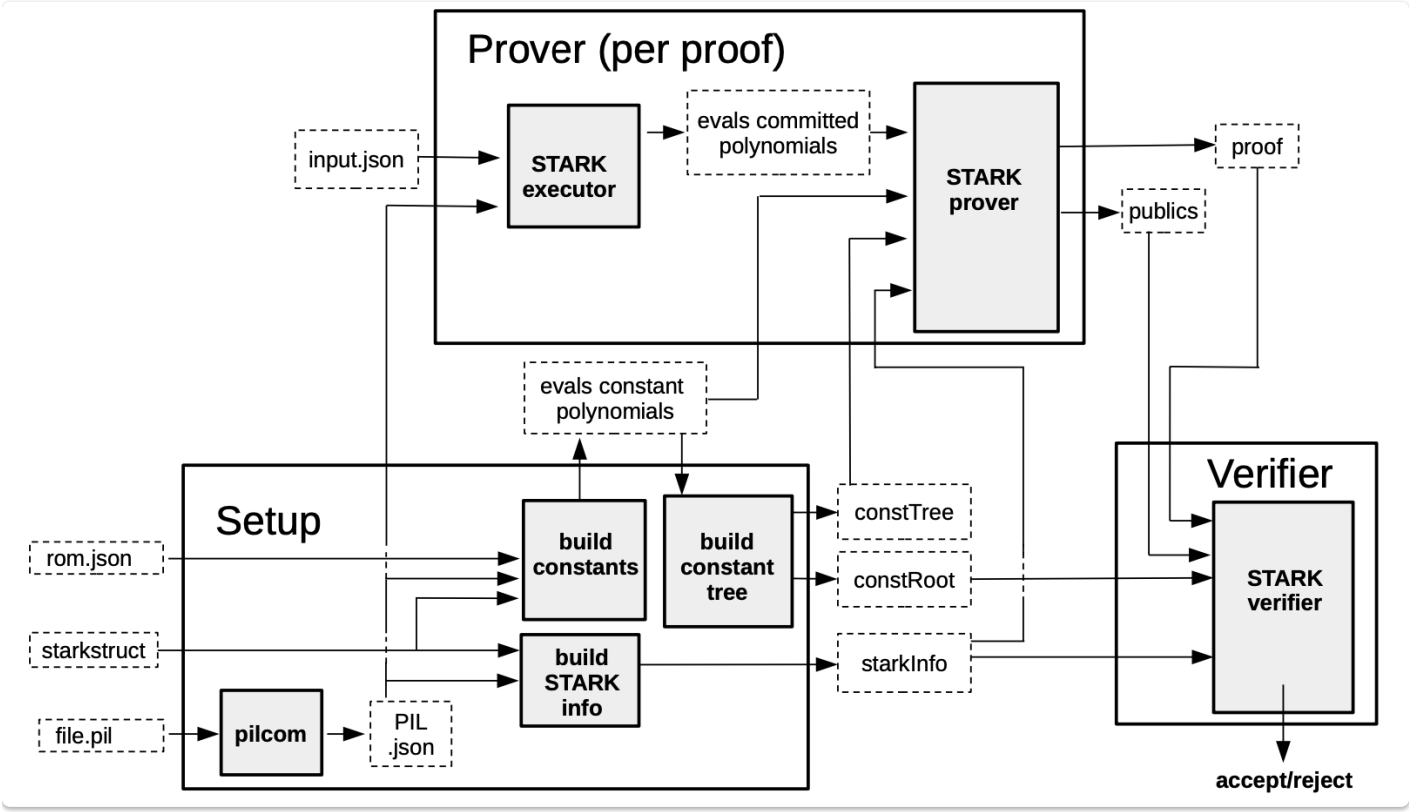
4. Constant-round polynomial IOP + KZG polynomial commitment
   For example : Marlin-KZG, PlonK-KZG
   Pros: Universal trusted setup
   Cons: Proofs are ~4x-6x larger than Groth16, P is slower than Groth16, also not post-quantum.

Polygon zkEVM Proving system

See Docs

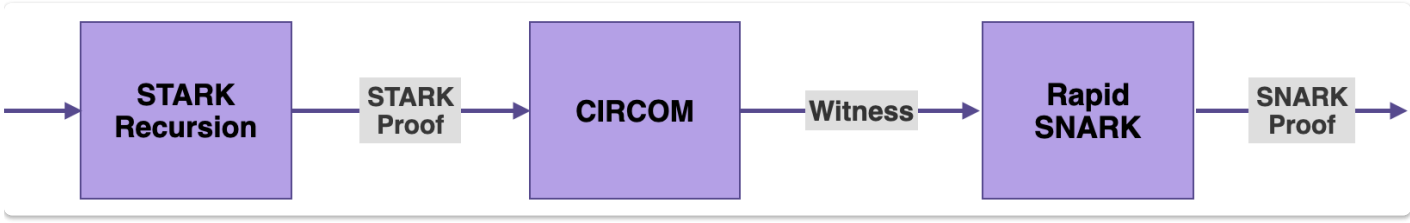
Polygon zkEVM Design approach





The general approach to designing the zkProver so as to realise a proof system based on State Machines is as follows:

1. Turn the required deterministic computation into a state machine computation.
2. Describe state transitions in terms of algebraic constraints. These are like rules that every state transition must satisfy.
3. Use interpolation of state values to build polynomials that describe the state machine.
4. Define polynomial identities that all state values must satisfy.
5. A specially designed cryptographic proving system (e.g. a STARK, a SNARK, or a combination of the two) is used to produce a verifiable proof, which anyone can verify.



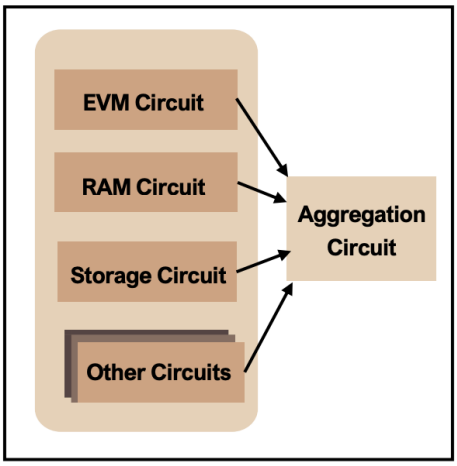
Scroll Backend

From Scroll [presentation]

Two-layer architecture

Scroll

zkEVM



- The second layer needs to be verifier efficient (in EVM)
  - Proof is efficiently verifiable on EVM (small proof, low gas cost)
  - Prove the verification circuit of the former layer efficiently
  - Ideally, hardware friendly prover
  - Ideally, transparent or universal trusted setup
- Some promising candidates
  - Groth16
  - Plonk with very few columns
    - KZG/Fflonk/Keccak FRI (larger code rate)

Polygon Proof composition, recursion and aggregation

See documentation

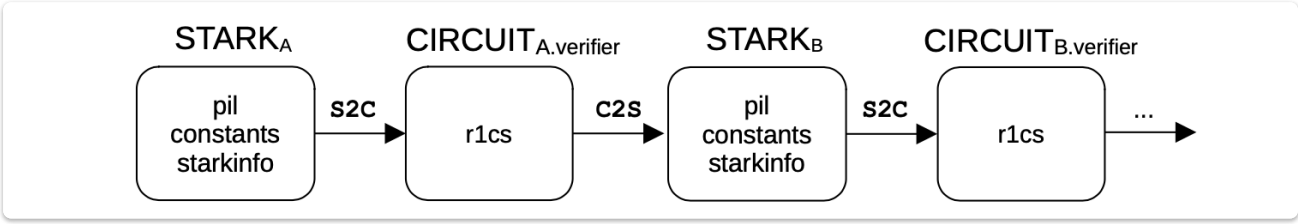
Composition

We have already seen that Polygon composes proofs of different types to take advantage of the different types of proving systems.

Recursion

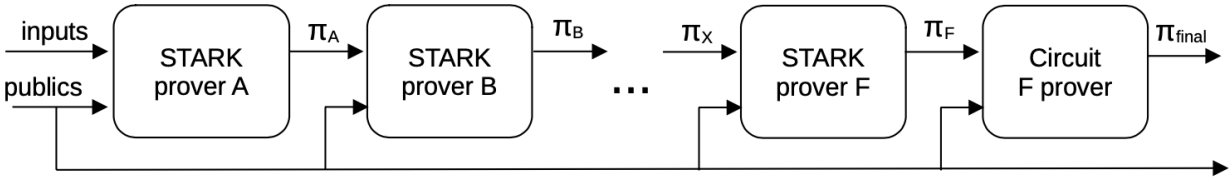
Proof recursion is a powerful technique.
In Polygon this is implemented in 2 phases

1. Setup phase.
   "The design idea is to create a cascade of Verifier circuits, where at each step a lot more efficiently verifiable proof is recursively created. The process therefore consists of an alternating series of two sub-processes; converting a STARK proof into a Verifier circuit, and converting a Verifier circuit into a STARK proof, denoted by S2C and C2S, respectively."



2. Proving phase
   The first proof is generated by providing the first STARK Prover with the proper inputs and public values. The output proof is then passed as input to the

next STARK Prover, together with public inputs. This process is recursively repeated.



Aggregation

Aggregation is a type of proof composition in which multiple valid proofs can be collated and proved to be valid by using one proof, called the Aggregated Proof. Validating such an Aggregated Proof is equivalent to validating all the collated proofs.