Lesson 11 - Risc Zero / PLONK

Risc Zero

Introduction

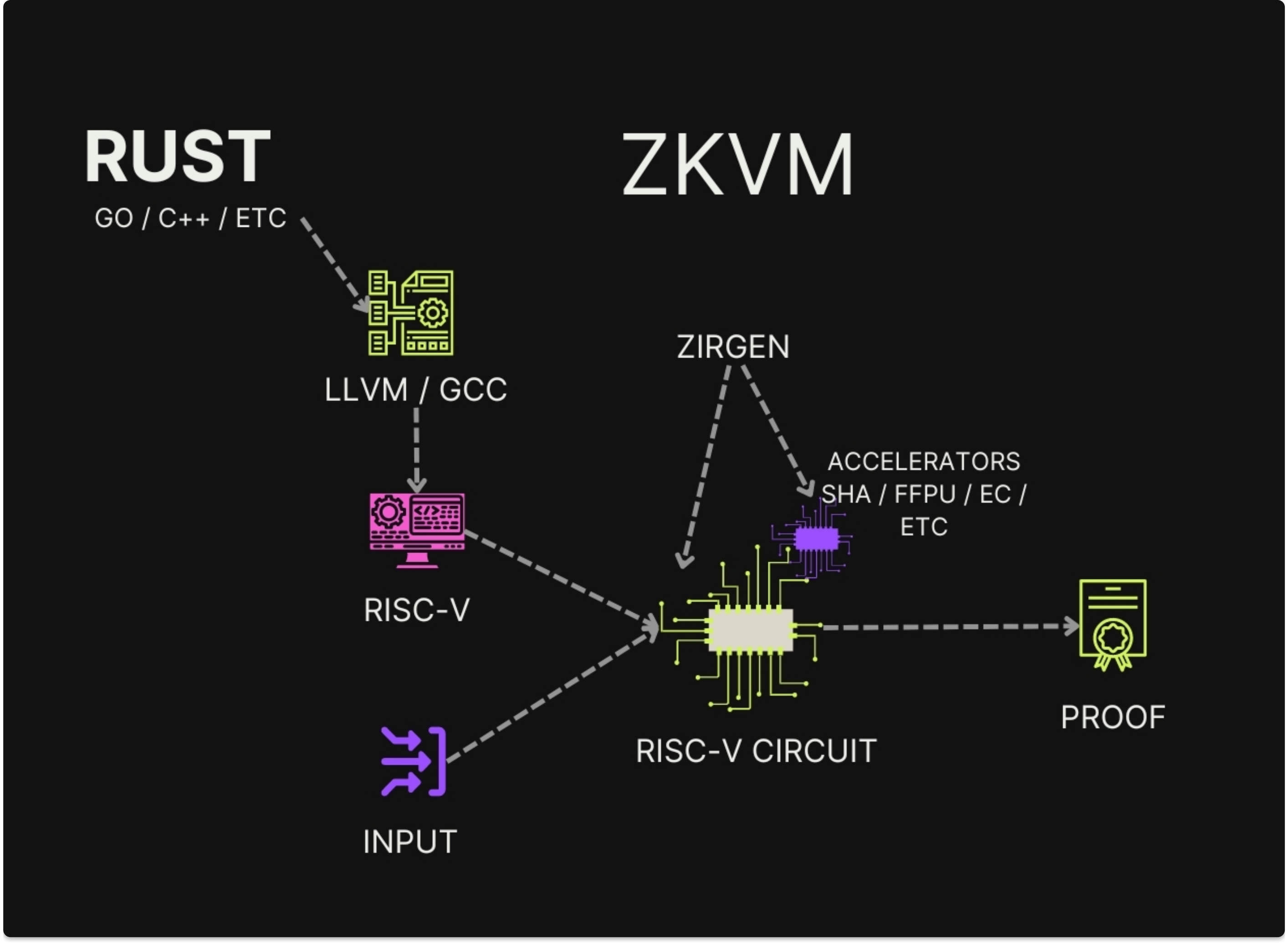
The RISC Zero zkVM is an open-source, zero-knowledge virtual machine designed for constructing trustless, verifiable software applications.

Risc Zero's goal is to integrate existing programming languages and developer tools into the zero-knowledge realm. This is accomplished through a high-performance ZKP prover, which provides the performance allowance necessary to create a zero-knowledge virtual machine (zkVM) implementing a standard RISC-V instruction set.

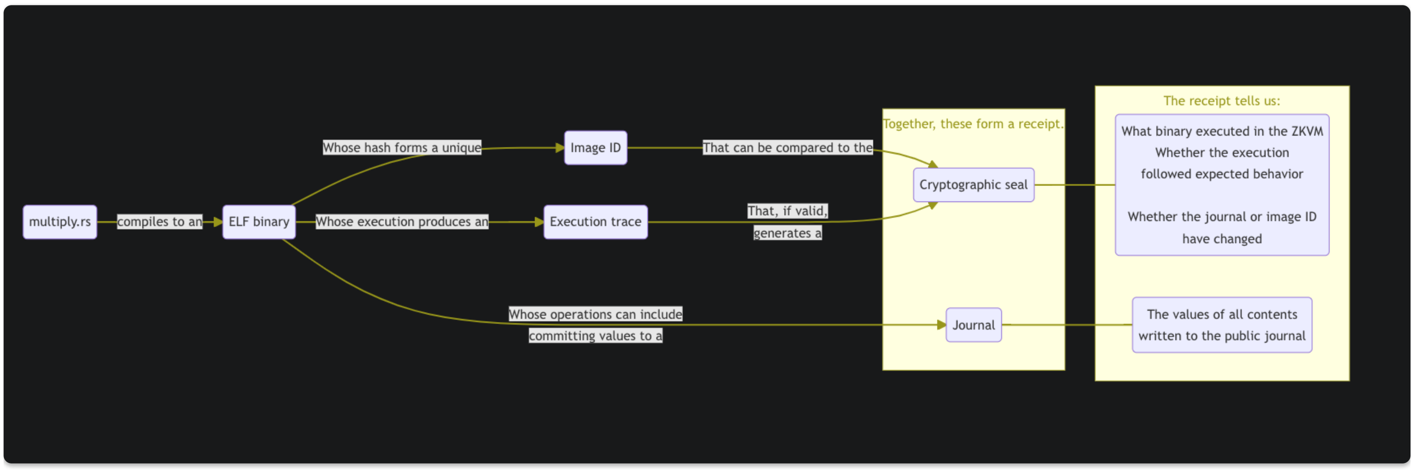
In practical terms, this allows for smooth integration between "host" application code, written in high-level languages running natively on host processors (e.g., Rust on arm64 Mac), and "guest" code in the same language executing within the zkVM.

A zero-knowledge virtual machine is a virtual machine that runs trusted code and generates proofs that authenticate the zkVM output.
RISC Zero's zkVM implementation, based on the RISC-V architecture, executes code and produces a computational receipt.

Risc-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles


RUST
GO / C++ / ETC
ZKVM
LLVM / GCC
ZIRGEN
ACCELERATORS
SHA / FFPU / EC /
ETC
RISC-V
RISC-V CIRCUIT
PROOF
INPUT

In a RISC Zero zkVM program, guest code written for the zkVM is compiled to an ELF binary and executed by the `prover`, which returns a computational receipt to the host program
Anyone possessing a copy of this receipt can verify the program's execution and access its publicly shared outputs.



multiply.rs
compiles to an
ELF binary
Whose hash forms a unique
Image ID
That can be compared to the
Whose execution produces an
Execution trace
That, if valid, generates a
Whose operations can include committing values to a
Together, these form a receipt.
Cryptographic seal
Journal
The receipt tells us:
What binary executed in the ZKVM
Whether the execution followed expected behavior
Whether the journal or image ID have changed
The values of all contents written to the public journal

Before being executed on the zkVM, guest source code is converted into a RISC-V ELF binary.
The binary file is hashed to create a `image ID` that uniquely identifies the binary being executed.
The binary may include code instructions to publicly commit a value to the `journal`. Later, the journal contents can be read by anyone with the receipt.

After the binary is executed, an execution trace contains a complete record of zkVM operation.
The trace is inspected and the ELF file's instructions are compared to the operations that were actually performed.
A valid trace means that the ELF file was faithfully executed according to the rules of the RISC-V instruction set architecture.

The execution trace and the journal are then used to generate a seal, a blob of cryptographic data that shows the receipt is valid.
The seal has properties that reveal whether itself or the journal have been altered.
When the receipt is verified, the seal will be checked to confirm the validity of the receipt.
To check whether the correct binary was executed, the seal can be compared to the image ID of the expected ELF file.

Proof System

When a the RISC Zero zkVM executes, it produces a computational receipt that consists of:

- a journal which contains the public outputs of the computation, and
- a seal which is a zkSTARK

Given a `receipt` and an `Image ID` a skeptical third party can verify the purported output of the computation.

Further details

This includes

- Code to emulate RISC-V, including decoding RISC-V instructions and constructing the execution trace.
- Code to evaluate the constraint polynomials that check the execution trace.

The image ID is the SHA-2 hash of the image of the initial zkVM memory state.

Checking the Image ID

The image ID can be determined from the compiled ELF source code. Someone wishing to confirm that a receipt corresponds to Rust source code can compile that code targeting the RISC Zero zkVM and verify that the image ID resulting from this compilation matches the image ID in the receipt.

Like other zk-STARKs, RISC Zero's implementation makes it cryptographically infeasible to generate an invalid receipt:

- If the binary is modified, then the receipt's seal will not match the image ID of the expected binary.
- If the execution is modified, then the execution trace will be invalid.
- If the output is modified, then the journal's hash will not match the hash recorded in the receipt.

Once the proof receipt has been generated it can be sent via side channel to the verifier.
The verifier does not need access to the host code, but they do need the image ID.

Structure of a zkVM program

In typical use cases, a RISC Zero zkVM program will actually be structured with three components:

- Source code for the *guest*,
- Code that *builds* the guest's source code into executable methods, and
- Source code for the *host*, which will call these built methods.

The code for each of these components uses its own associated RISC Zero crate or module:

- The *guest* code uses the `guest` module of the `risc0-zkvm` crate
- The *build* code for building guest methods uses the `risc0-build` crate
- The *host* code uses the `risc0-zkvm` crate

Example program

In the [repo](https://github.com/ExtropyIO/ZeroKnowledgeBootcamp/tree/main/risc0/examples, we have 3 example programs.

Password checker

Overall process

Alice wants to create a password, that complies with Bob's password requirements without revealing her password.
The program is divided between a host driver that runs the zkVM code and a guest program that executes on the zkVM.
Alice can run a password validity check and her password never needs to leave her local machine.
The process is as follows :

- Alice's `host driver program` shares a password and salt with the `guest zkVM` and initiates the guest program execution.
- The `guest zkVM program` checks Alice's password against a set of validity requirements, such as minimum length, inclusion of special characters.
- If the password is valid, it is hashed with the provided salt using SHA-256. If not, the program panics and no computational receipt is generated.
- The guest program generates a salted hash of Alice's password and commits it to a `journal`, part of a computational `receipt`.
- Alice sends the receipt to Bob's Identity Service.

The `image ID` and the `journal` on the receipt provide Bob assurance that:

The program Alice executed within the zkVM was actually Bob's Password Checker, and Bob's Password Checker approved Alice's password

It demonstrates that the guest zkVM program has executed, which tells Bob `his password requirements were met`.
It also provides a tamper-proof `journal` for public outputs, the integrity of which tells Bob that **the shared outputs are the result of running the password program**.

Application design

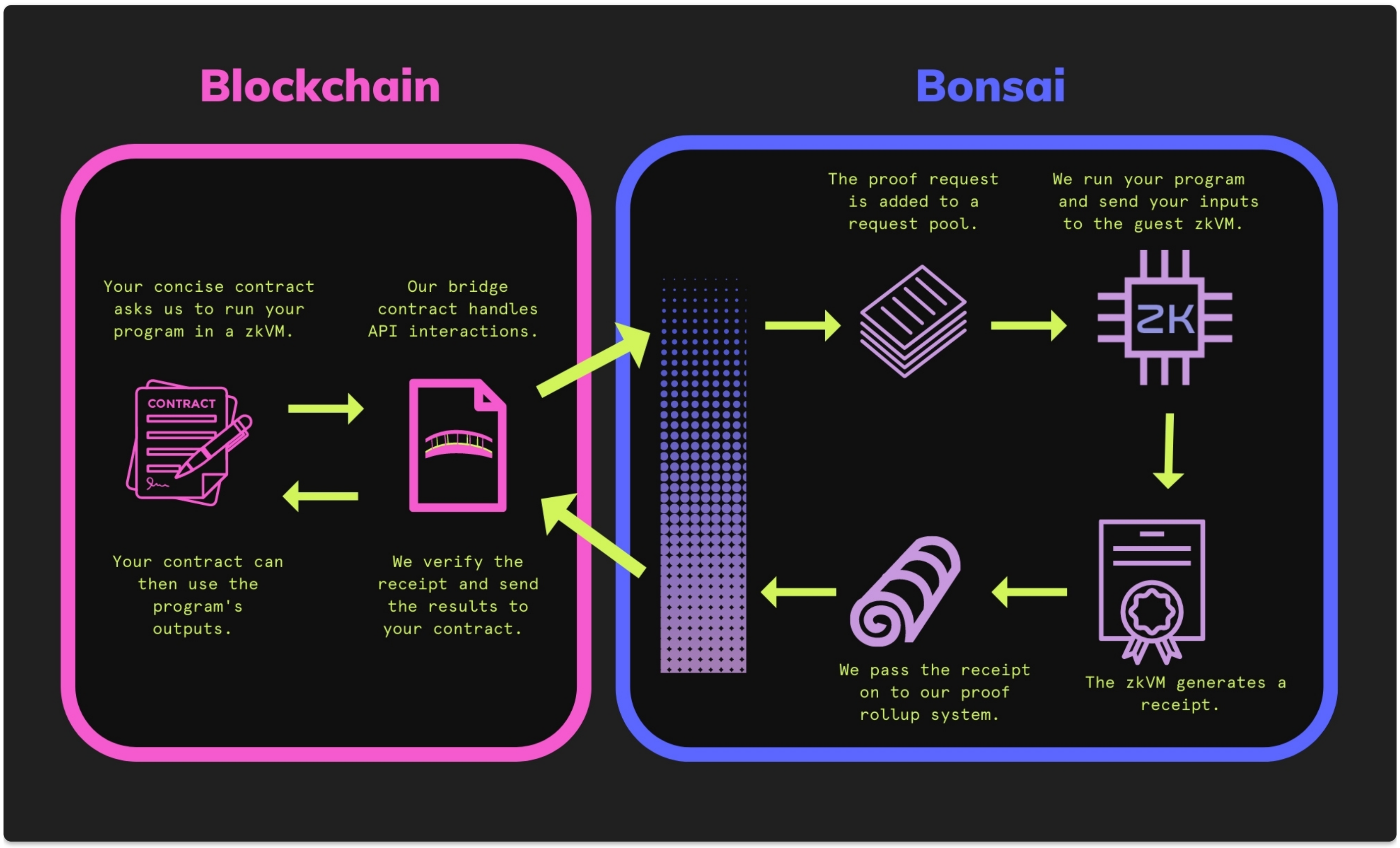
We split the code between guest and host, if there is a part of the computation that doesn't need to be performed securely, then it can be run outside the zkVM.
For optimisation, when using cryptographic operations, it is possible to build ‘accelerator’ circuits such as the Risc zero implementation of SHA26.
Fast cryptography is sufficient to support many ‘DeFi’ applications. For many other applications, it is possible to perform most computation on the host (outside the zkVM) and then verify the results in the zkVM.
In terms of program size, although there is a theoretical maximum size is 128 MB, with the current implementation programs should be kept to at most ~ 1MB.

Proving the validity of two RISC Zero receipts currently takes ~10 seconds.

Rust compatibility

Risc zero are working to include the rust standard library, until that is complete it is advised to use the `no_std` option

Bonsai



Blockchain
Bonsai
Your concise contract asks us to run your program in a zkVM.
Our bridge contract handles API interactions.
CONTRACT
Your contract can then use the program's outputs.
We verify the receipt and send the results to your contract.
The proof request is added to a request pool.
We run your program and send your inputs to the guest zkVM.
ZK
The zkVM generates a receipt.
We pass the receipt on to our proof rollup system.

See lite paper
Bonsai is a versatile zero-knowledge proof network designed to facilitate the integration of ZK proofs for scalability, privacy, and other benefits across any chain, protocol, or application. This includes ETH, L1 Blockchains, Cosmos App Chains, L2 Rollups, and DApps.

The unique Bonsai network is built on three essential components, paving the way for the development of innovative applications in both blockchain and traditional application domains:

- A flexible zkVM that can operate any VM within a zero-knowledge/verifiable context
- A proving system that seamlessly connects to any smart contract or chain
- A universal rollup that distributes all computations proven on Bonsai across every chain

```
  // Without Bonsai
contract simulation_normal {
 function some_really_hard_work() {
   // A large amount of gas heavy code
       // code ...
       // code ...
       // code ...
       // code ...
       // code ...
       //
 }
}
```

```
  // With Bonsai
contract simulation_bonsai {
 function some_really_hard_work() {
   bonsai_proving_network.call(“some_really_hard_work”);
 }
}
```

Setting up a project

Install rust if you haven't already done so.
`curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`

Install the tool

```
## Install from crates.io
cargo install cargo-risczero
```
language-bash

Using the template

```
## Create a project from the main template
        cargo risczero new my_project
## Create a project with 'std' support in the guest
        cargo risczero new my_project --std
## Disable git initialization
        cargo risczero new my_project --no-git
## Create from github template
        cargo risczero new my_project --template
https://github.com/risc0/risc0-rust-starter
```
language-bash

Examples

Wordle

See Wordle repo

Sudoku

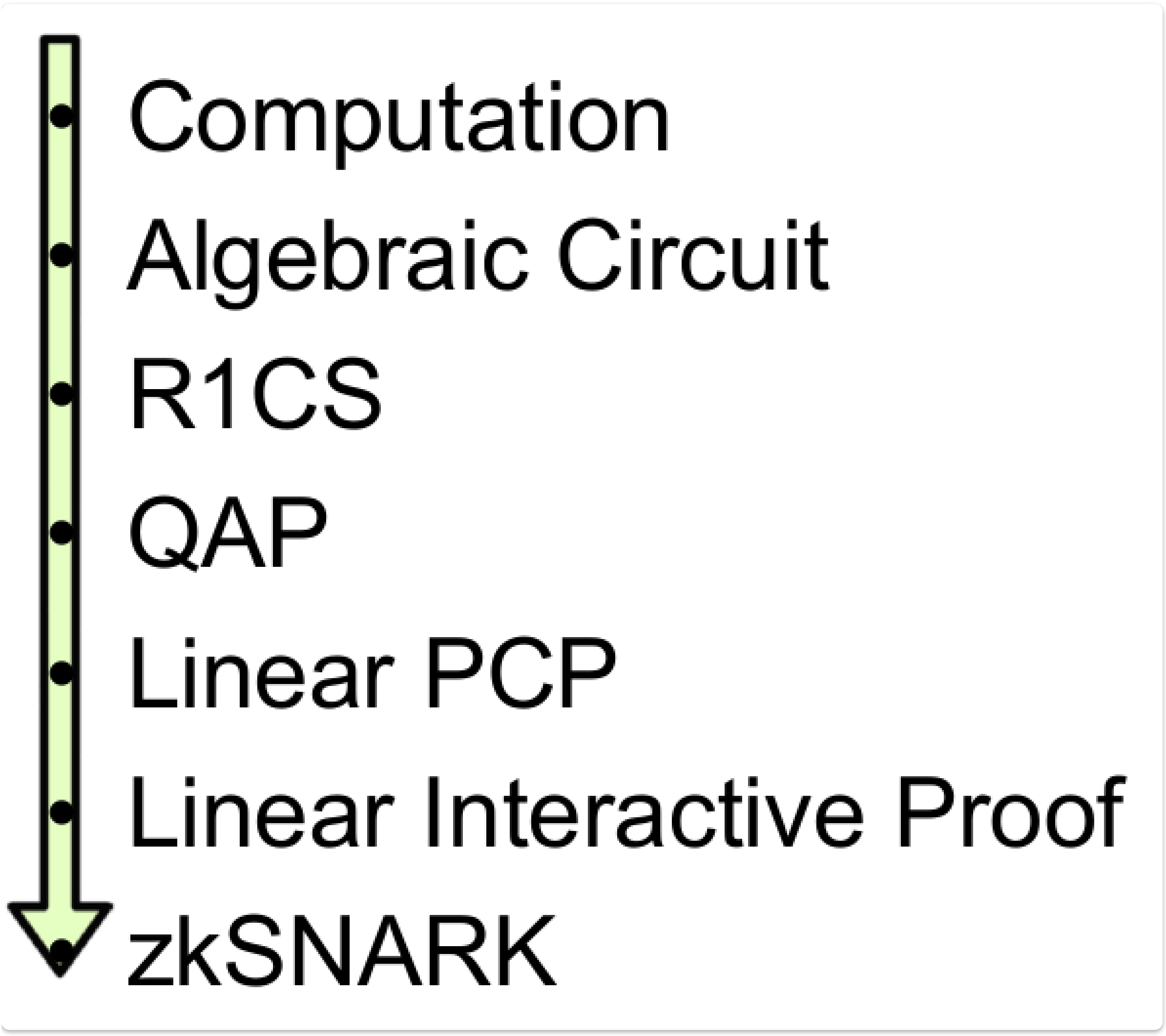
See Repo

ZKSnark Process

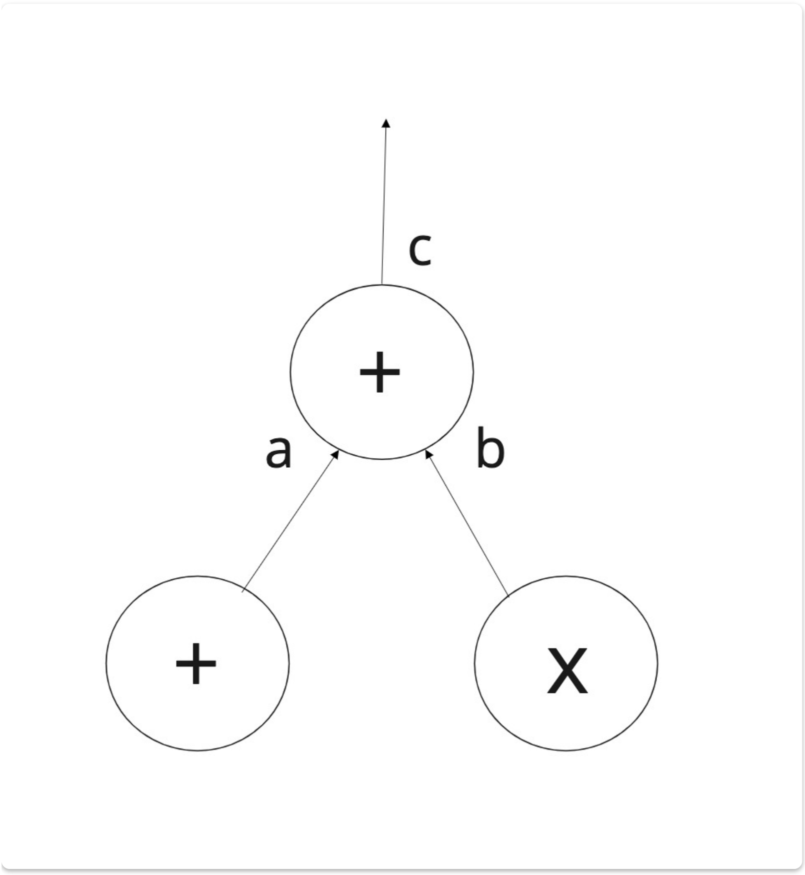
General Process

- Arithmetisation
    - Flatten code
    - Arithmetic Circuit
    - R1CS
    - QAP
- Polynomials
- Polynomial Commitment Scheme
- Inner product argument
- Cryptographic proving system
- Make non interactive

Transformations in SNARKS



1. Trusted Setup
   ZKSNarks require a one off set up step to produce prover and verifier keys. This step is generally seen as a drawback to zkSNARKS, it requires an amount of trust, if details of the setup are later leaked it would be possible to create false proofs.
2. A High Level description is turned into an arithmetic circuit
   The creator of the zkSNARK uses a high level language to specify the algorithm that constitutes and tests the proof.
   This high level specification is compiled into an arithmetic circuit.
   An arithmetic circuit can be thought of as similar to a physical electrical circuit consisting of logical gates and wires. This circuit constrains the allowed inputs that will lead to a correct proof.



3. Further Mathematical refinement
   The circuit is then turned into a an R1CS, and then a series of formulae called a Quadratic Arithmetic Program (QAP).
   The QAP is then further refined to ensure the privacy aspect of the process.
   The end result is a proof in the form of series of bytes that is given to the verifier. The verifier can pass this proof through a verifier function to receive a true or false result.
   There is no information in the proof that the verifier can use to learn any further information about the prover or their witness.

Trusted Setups

From ZCash explanation :

"SNARKs require something called “the public parameters”. The SNARK public parameters are numbers with a specific cryptographic structure that are known to all of the participants in the system. They are baked into the protocol and the software from the beginning.

The obvious way to construct SNARK public parameters is just to have someone generate a public/private keypair, similar to an ECDSA keypair, (See ZCash explanation) and then destroy the private key.

The problem is that private key. Anybody who gets a copy of it can use it to counterfeit money. (However, it cannot violate any user’s privacy — the privacy of transactions is not at risk from this.)"

ZCash used a *secure multiparty computation* in which multiple people each generate a “shard” of the public/private keypair, then they each destroy their shard of the toxic waste private key, and then they all bring together their shards of the public key to to form the SNARK public parameters.
If that process works — i.e. if *at least one of the participants* successfully destroys their private key shard — then the toxic waste byproduct never comes into existence at all.

They have recently introduced Halo2 which eliminates the need for a trusted setup

Halo2 uses the curves Pallas and Vesta (collectively Pasta) which are also used by Mina

Pallas: y^2 = x^3 + 5y2=x3+5 over
GF(0x40000000000000000000000000000000224698fc094cf91b992d30ed00000001)

Vesta: y^2 = x^3 + 5y2=x3+5 over
GF(0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001)

The use “nested amortization”— repeatedly solving over cycles of elliptic curves so that computational proofs can be used to reason about themselves efficiently, which eliminates the need for a trusted setup.

Transforming our problem into a QAP

Lets look first at transforming the problem into a QAP, there are
3 steps :

- code flattening,
- conversion to a rank-1 constraint system (R1CS)
- formulation of the QAP.

Code Flattening

We are aiming to create arithmetic and / or boolean circuits from our code, so we change the high level language into a sequence of statements that are of two forms

x = y (where y can be a variable or a number)
and
x = y (op) z
(where op can be +, -, $*$, / and y and z can be variables, numbers or themselves sub-expressions).

For example we go from

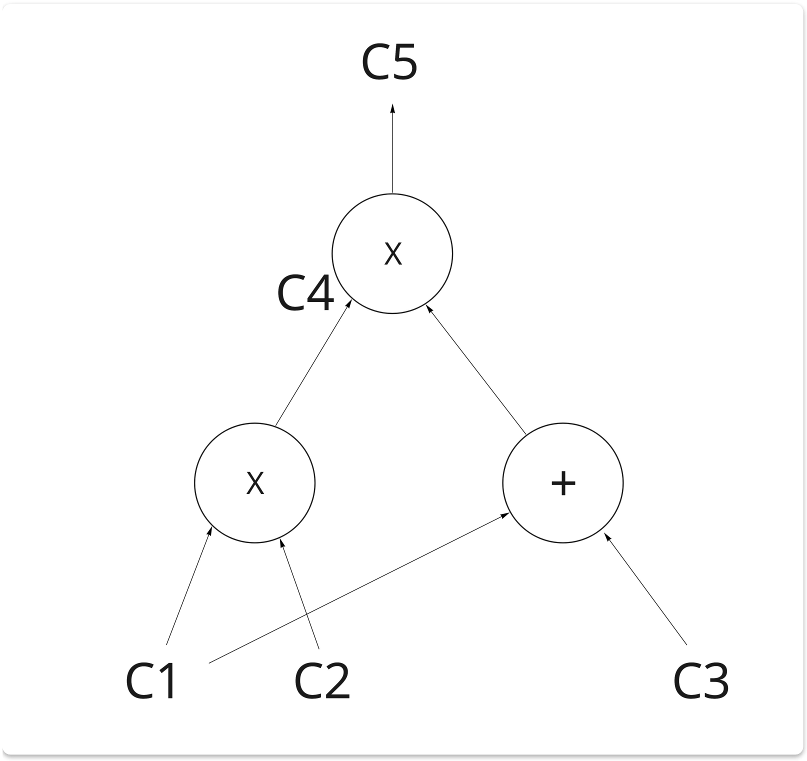
```
def qeval(x):
  y = x**3
  return x + y + 5
```

to

```
  sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

Arithmetic Circuit

This is a collection of multiplication and addition gates



Rank 1 Constraint Systems

Constraint languages can be viewed as a generalization of
functional languages:

- everything is referentially transparent and side-effect free
- there is no ordering of constraints
- composing two R1CS programs just means that their constraints are simultaneously satisfied.

(From http://coders-errand.com/constraint-systems-for-zk-snarks/)

The important thing to understand is that a R1CS is not a computer program, you are not asking it to produce a value from certain inputs. Instead, a R1CS is more of a verifier, it shows that an already complete computation is correct .

The arithmetc circuit is a compositon of multplicatve sub-circuits (a single multiplcation gate and mutiple addition gates)

A rank 1 constraint system is a set of these sub-circuits expressed as constraints, each of the form:
$AXB = C$
where A, B, C are each linear combinations c1· v1+ c2· v2+ ...
The c_i are constant field elements, and the v_i are instance or witness variables (or 1).

- $AXB = C$ doesn't mean C is computed from A and B just that A, B, C are consistent.

More generally, an implementation of $x = f(a, b)$ doesn't mean that x is computed from a and b, just that x, a, and b are consistent.

Thus our R1CS contains :

- the constant 1
- all public inputs
- outputs of the function
- private inputs
- auxilliary variables

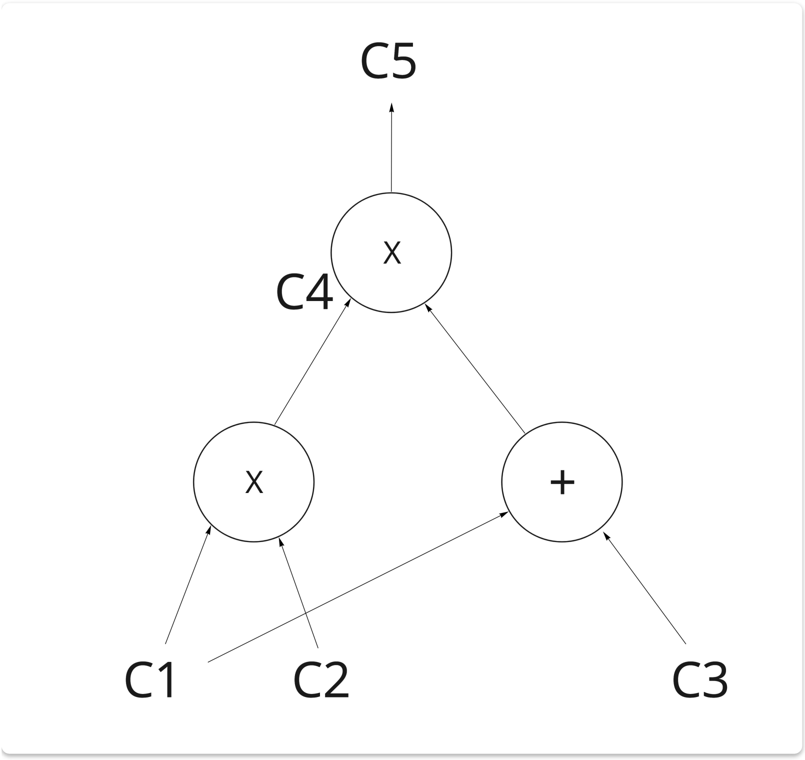
The R1CS has

- one constraint per gate;
- one constraint per circuit output.

Example

Assume Peggy wants to prove to Victor that she knows $c1, c2, c3$ such that $(c1 \cdot c2) \cdot (c1 + c3) = 7$

We transform the expression above into an arithmetic circuit as depicted below



A legal assignment for the circuit is of the form:

(c1, . . . , c5), where c4 = c1 · c2 and c5 = c4 · (c1 + c3).

Plonkish protocols

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

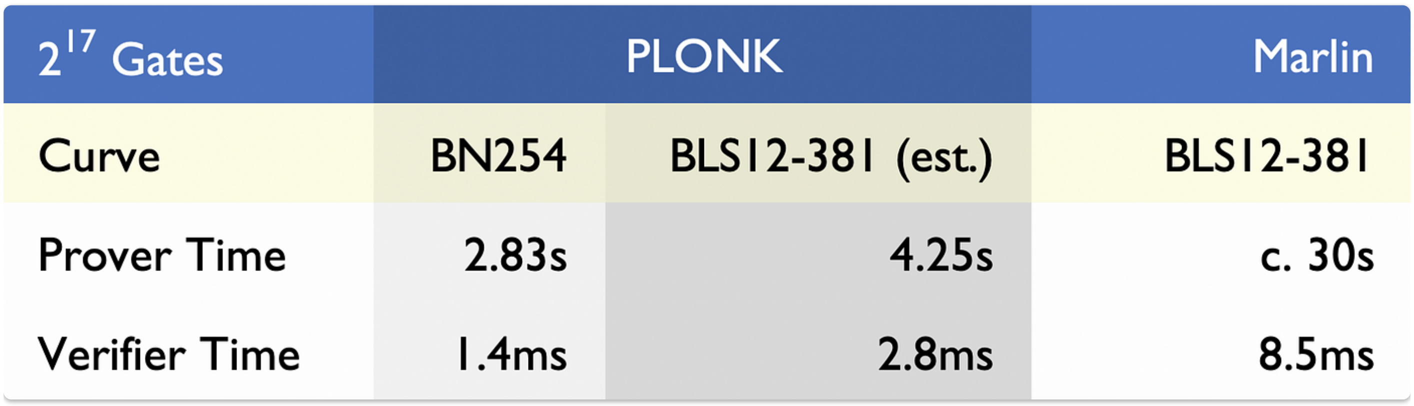
Before PLONK

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 $G, G \cdot s, G \cdot s^2 \dots G \cdot s^n$, 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

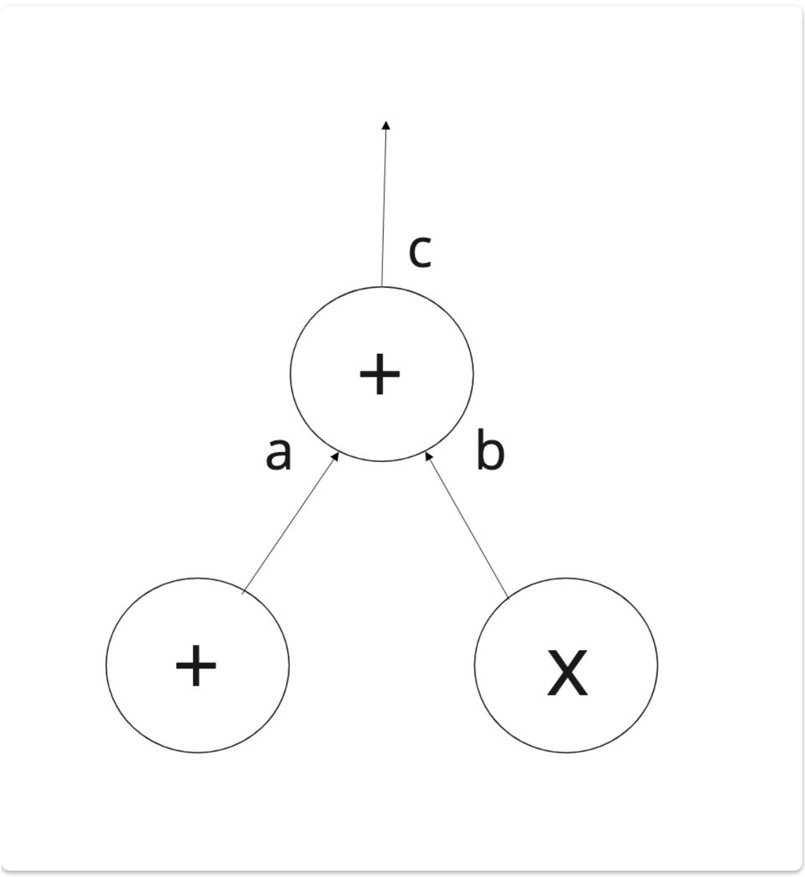
See PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge
Also see Understanding PLONK

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:

Arithmetic Circuits



Developing a circuit

Once we have a (potentially large)circuit, we want to get it into a more usable form, so we can put the values into a table detailing the inputs and outputs for a gate.

So for gates 1 to i we can represent the a, b and c values as
a_i, b_i, c_i

And if the circuit is correct then for an addition gate

a_i+ b_i = c_i

or
a_i+ b_i - c_i = 0

and for a multiplication gate

$a_i.b_i$ - c_i = 0

We would end up with a table like this

	a	b	c	S
1	a_1	a_1	a_1	1
2	a_2	b_2	c_2	1
3	a_3	b_3	c_3	0

But we would also want to know what type of gate it is, there is a useful trick where we introduce a selector S, which is 1 for an addition gate and 0 for a multiplication gate.

We can then generalise our equation as

$S_i(a_i + b_i) + (1 - S_i)a_i.b_i - c_i = 0$

These are called the *gate constraints* because they refer to the equalities for a particular gate.

We can also have *copy constraints* where we have a relationship between values that are not on the same gate, for example it may be the case that

a_7 = b_5 for a particular circuit, in fact this is how we link the gates together.

In PLONK we also have constant gates and more specialised gates.
For example representing a hash function as a series of generic gates (addition, multiplication and constant) would be inefficient.

From Zac Williamson

"PLONK's strength is these things we call custom gates. It's basically you can define your own custom bit arithmetic operations. I guess you can call them mini gadgets.
That are extremely efficient to evaluate inside a PLONK circuit.
So you can do things like do elliptical curve point addition in a gate.

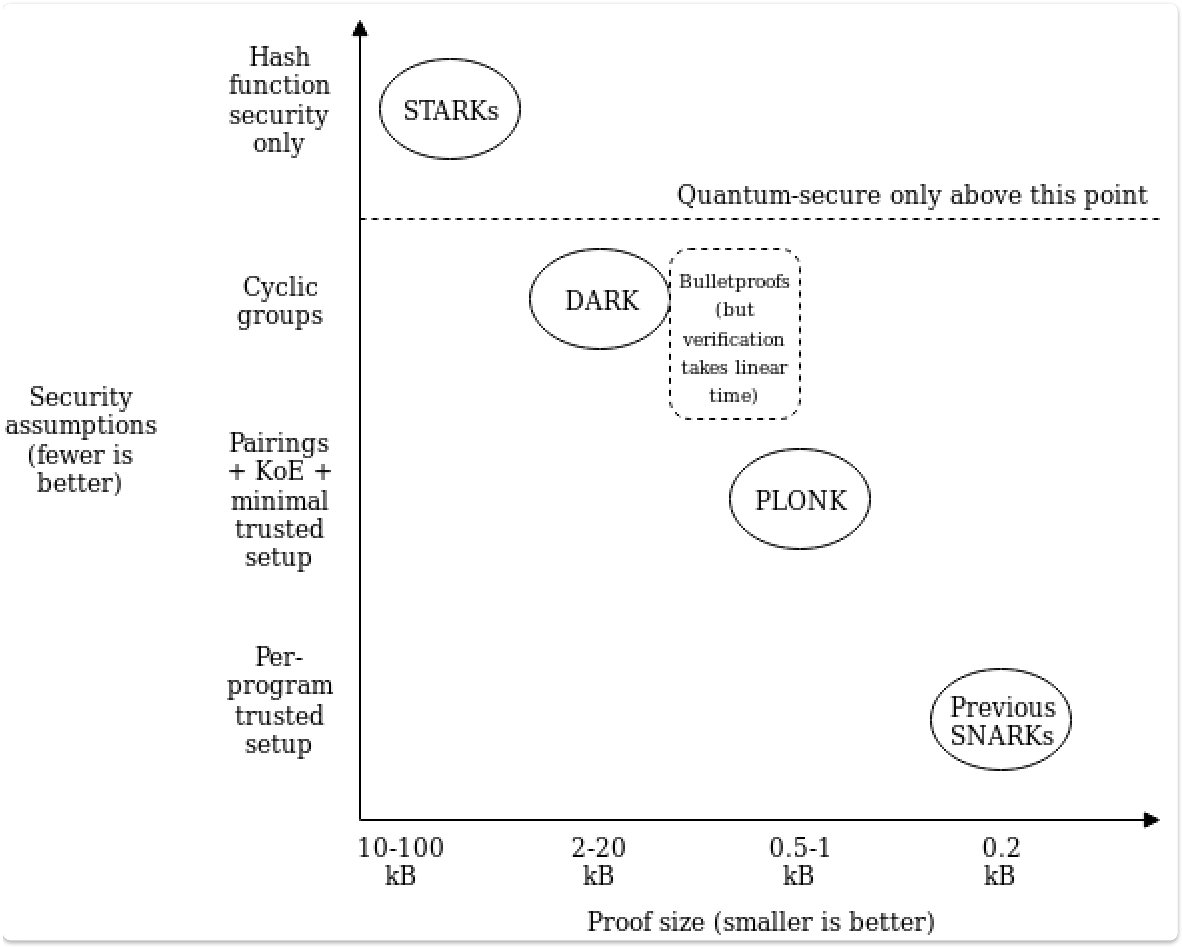
> You can do things like efficient Poseidon hashes, Pedersen hashes. You can do parts of a SHA-256 hash. You can do things like 8-bit logical XOR operations.
> All these like explicit instructions which are needed for real world circuits, but they’re all quite custom."

Plookup

Some operations such as boolean operations do not need to be computed, but can be put into a lookup table. Similarly public data can be put in a lookup table.
For example we could use a lookup table for the XOR operation, or include common values as we saw in the range proofs in Aztec.

Polynomial Commitments in PLONK

PLONK uses Kate commitments based on trusted setup and elliptic curve pairings, but these can be swapped out with other schemes, such as FRI (which would turn PLONK into a kind of STARK



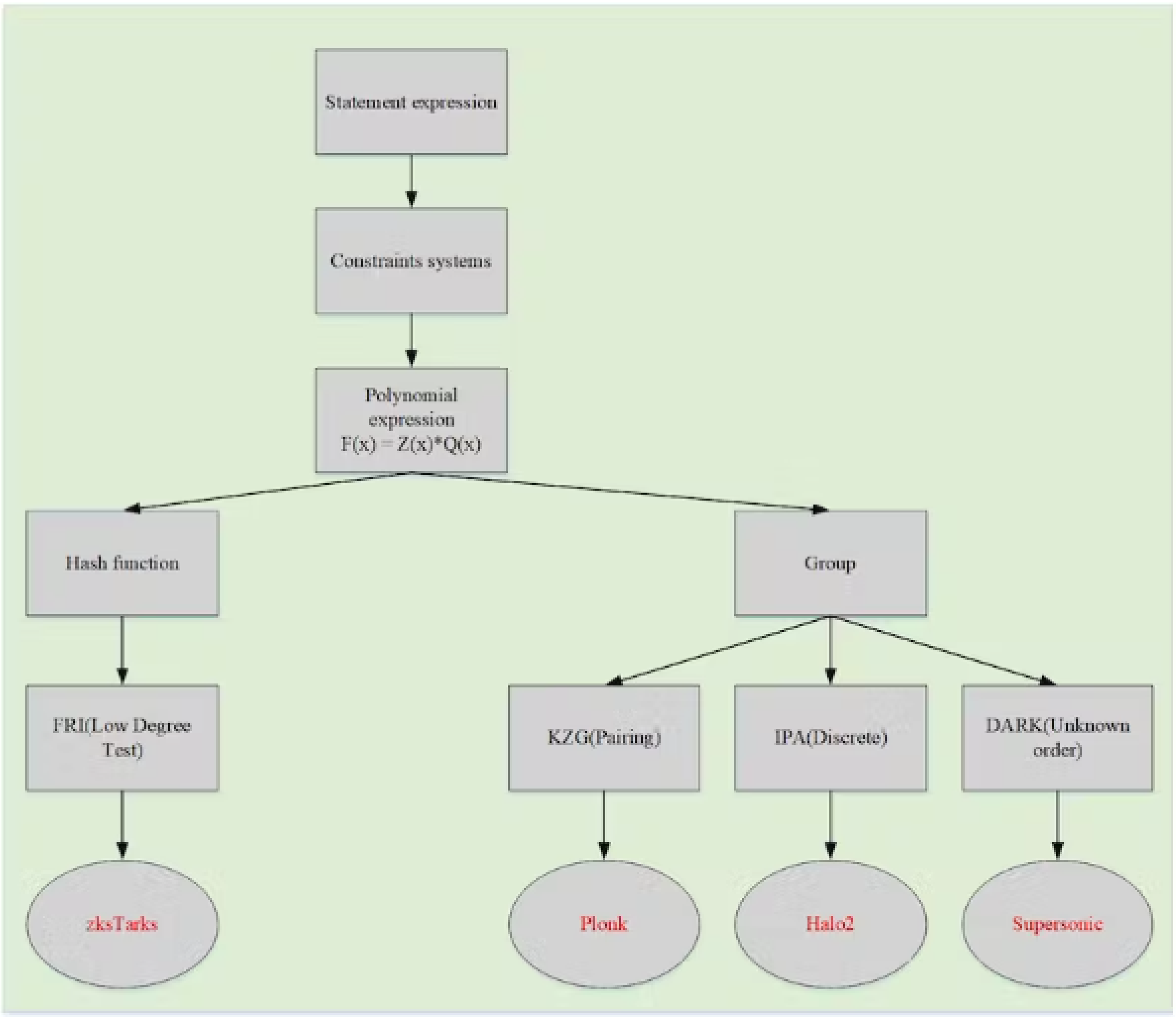
This means the arithmetisation - the process for converting a program into a set of polynomial equations can be the same in a number of schemes.
If this kind of scheme becomes widely adopted, we can thus expect rapid progress in improving shared arithmetisation techniques.

For a detailed talk about some of the custom gates used in Plonk see this video

For more information about the KZG commitment scheme see this introduction

Other commitment schemes are available :
Dory
Dark

For an analysis of polynomial commitment schemes, see this article


Statement expression
Constraints systems
Polynomial expression F(x) = Z(x)*Q(x)
Hash function
Group
FRI(Low Degree Test)
KZG(Pairing)
IPA(Discrete)
DARK(Unknown order)
zksTarks
Plonk
Halo2
Supersonic

Plonky2

From article

Plonky2 also allows us to speed up proving times for proofs that don't involve recursion. With FRI, you can either have fast proofs that are big (so they're more expensive to verify on Ethereum), or you can have slow proofs that are small. Constructions that use FRI, like the STARKs that Starkware uses in their ZK-rollups, have to choose; they can't have maximally fast proving times and proof sizes that are small enough to reasonably verify on Ethereum.

Plonky2 eliminates this tradeoff. In cases where proving time matters, we can optimize for maximally fast proofs. When these proofs are recursively aggregated, we're left with a single proof that can be verified in a small circuit. At this point, we can optimize for proof size. We can shrink our proof sizes down to 45kb with only 20s of proving time (not a big deal since we only generate when we submit to Ethereum), dramatically reducing costs relative to Starkware.

Plonky2 is natively compatible with Ethereum. Plonky2 requires only keccak-256 to verify a proof. We've estimated that the gas cost to verify a plonky2 size-optimized proof on Ethereum will be approximately 1 million gas.

However, this cost is dominated by the CALLDATA costs to publish the proof on Ethereum. Since CALLDATA was repriced in EIP-4488, the verification cost of a plonky2 proof has dropped to between 170-200k gas, which could make it not only the fastest proving system, but also the cheapest to verify on Ethereum.