A Gentle Introduction to Zero Knowledge Languages

Kostas Ferles
CRO, Veridise

Welcome!

- First and foremost a big welcome!
  - Thank you for attending Veridise's ZK-focused Secureum Workshop
  - Congratulations to the winners of RACE-23
- We have an exciting week planned for you
  - Daily lectures from Veridise about ZK technology and our tooling
  - Guest lectures by industry leaders

About Veridise

- Veridise is a blockchain security company
  - Founded by a team of world-class researchers
  - **Our obsessions**: 1. reasoning about code 2. creating tools that help us find bugs or prove properties about code

Our team

About Veridise

- We performed audits for many ecosystems (e.g., Ethereum, NEAR, StarkWare) and for different kinds of use cases (e.g., AMMs, stablecoins, auctions, etc.)

Leader in auditing ZKP Circuits

MANTA NETWORK

Succinct

Scroll

==>circom CIRCUIT COMPILER

Semaphore

Trusted by leading projects

ankr

RIBBON

DOGECHAIN

Our audits

About Veridise

We are developing state-of-the-art automated security tooling

SaaS Platform

We will dive into two zk-related tools this week:
1. zk-Vanguard 2. Picus

Workshop Logistics

Few things before we begin!

- Each will have a Veridise lecture followed by a guest lecture
  - This will be followed by a quiz for RACE-23 winners

Schedule

Our discord

Veridise.

Guidelines for RACE-23 Winners

- We will use ***zk-secureum-private*** for general communication (e.g., quiz announcements)
- Any non answer-revealing question can be sent on ***zk-secureum-private***.
  - For general-interest questions use ***zk-secureum-public*** *(open to all)*
- If you are unsure about sending a message publicly, send it to your personal support channel (you'll be added by us)
- Discussing quiz answers on the public channel is not allowed!
- Quizzes deadlines are listed in the official schedule.
- The registration on SaaS must be done with the same e-mail address you provided Rajeev with.
- You'll receive an e-mail (if you haven't already) with a unique user ID. Don't share the user ID with anyone (except us)!

Veridise.

Prizes

1st

2k USDC

2nd 3rd

1k USDC

4th 5th

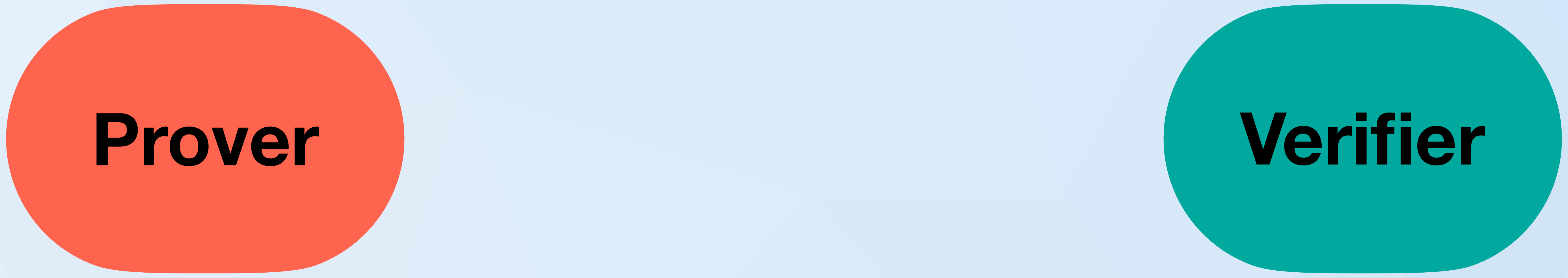
500 USDC

Top-performers will also be considered for an auditing position at Veridise :)

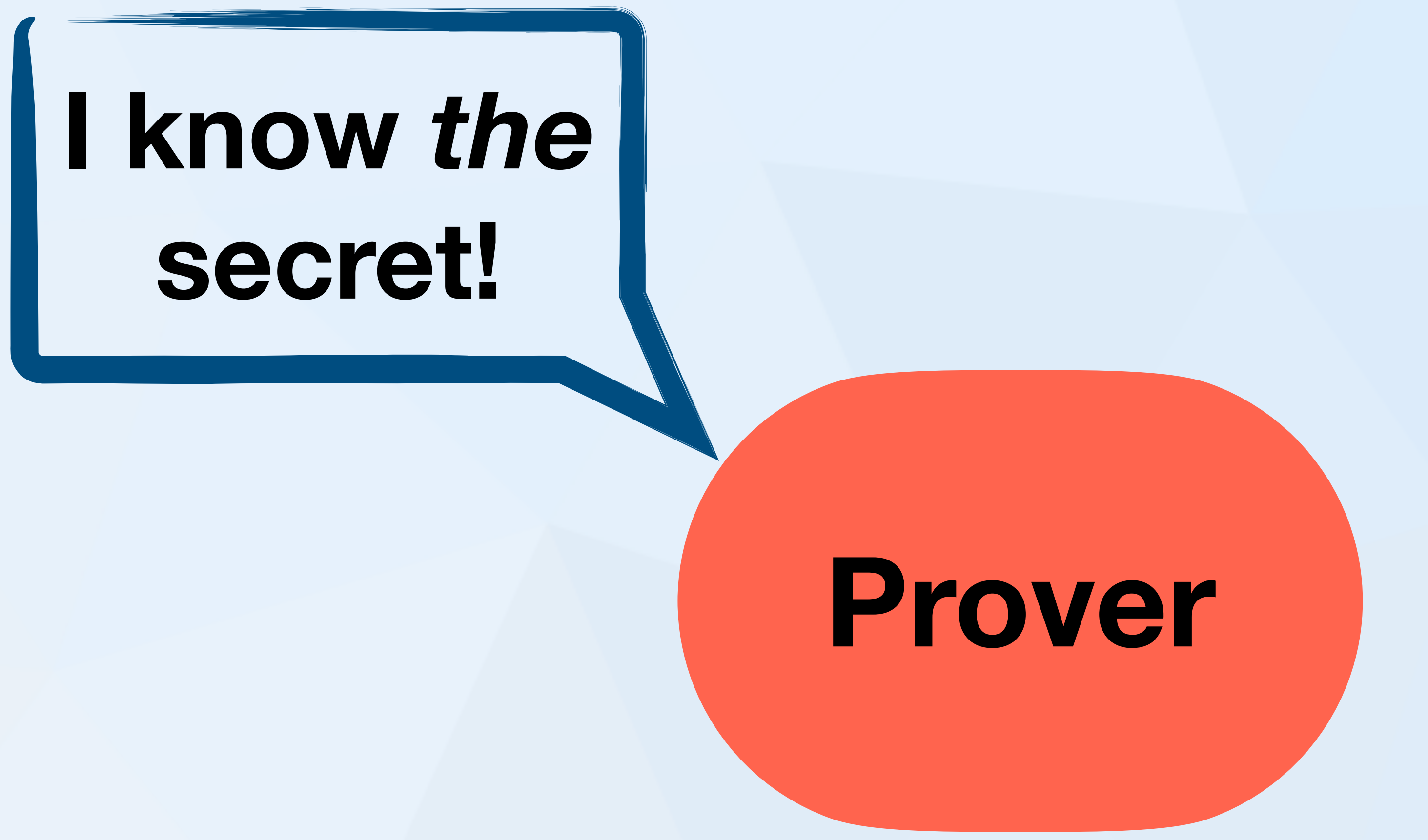
Let’s Cut to the Chase

An Introduction to
ZK Languages and Frameworks

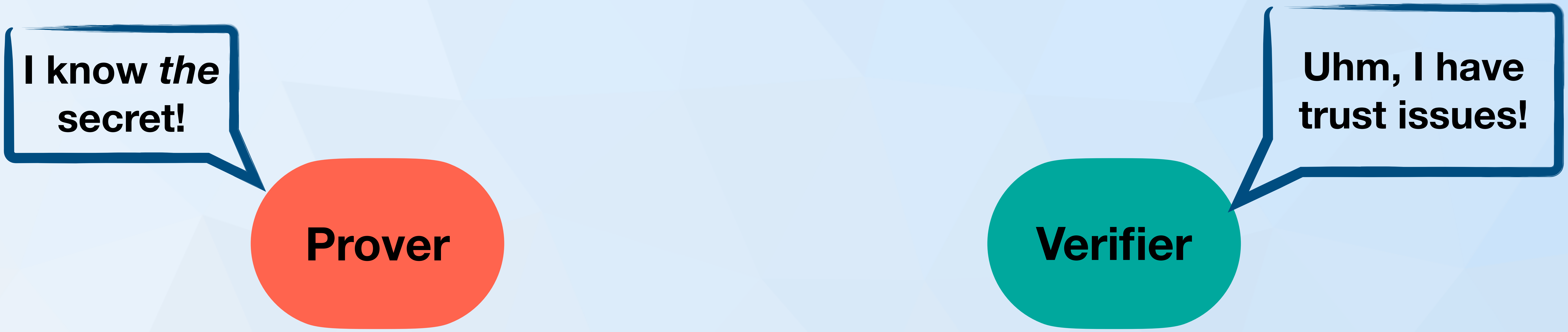
What is a ZK protocol?



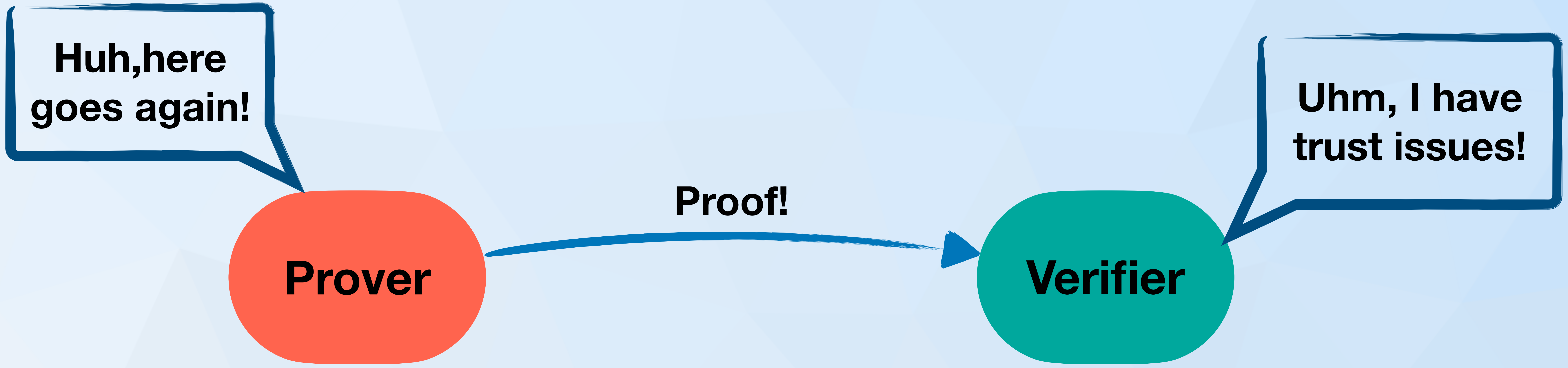
What is a ZK protocol?



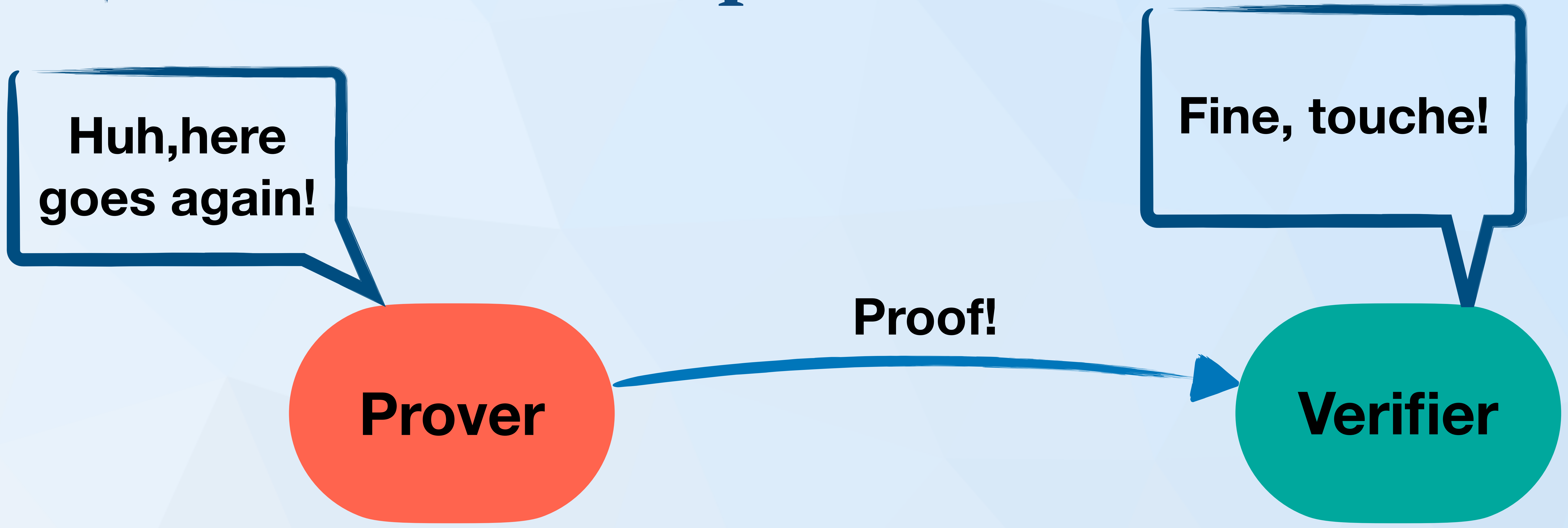
What is a ZK protocol?



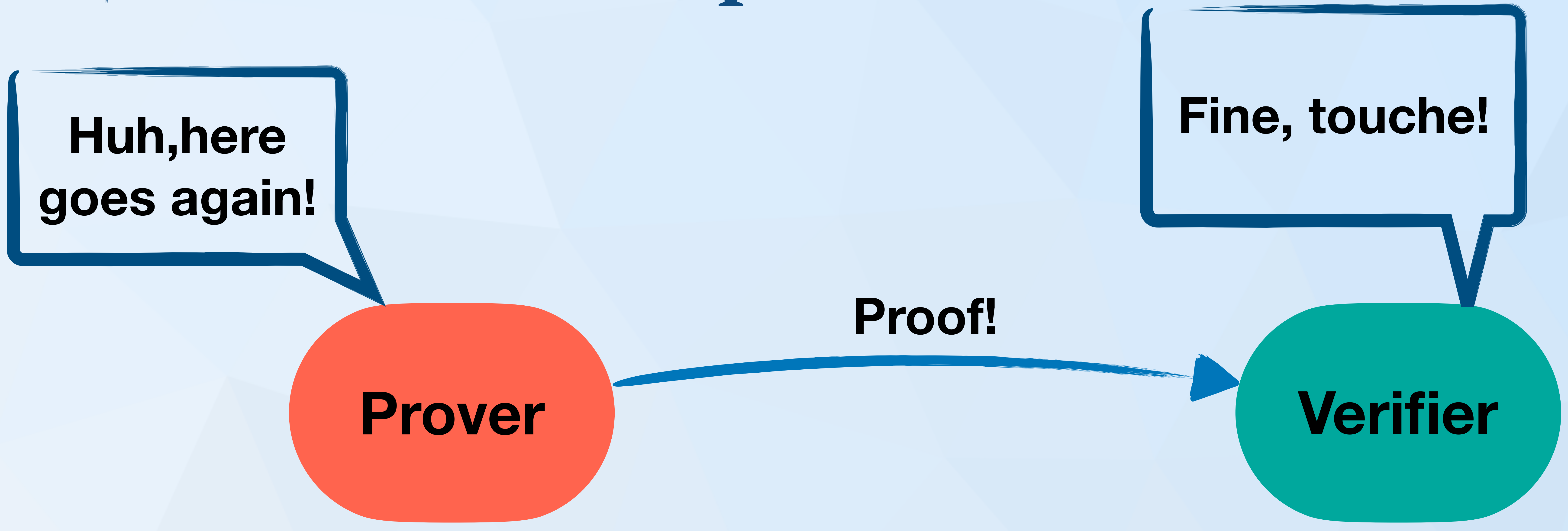
What is a ZK protocol?



What is a ZK protocol?



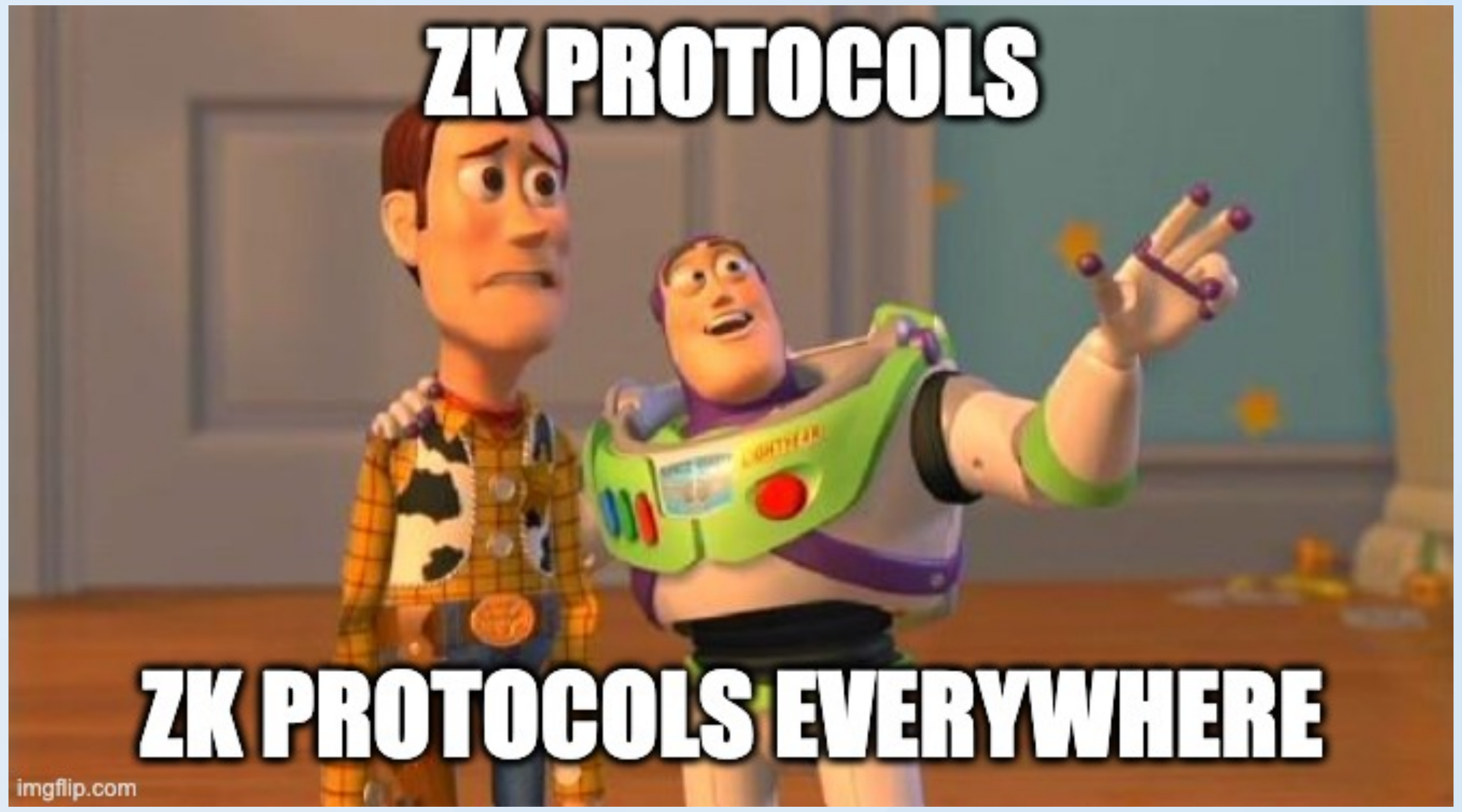
What is a ZK protocol?



- **Some *important* properties**
  - The prover doesn't reveal the secret
  - The verifier can always detect false proofs

The ZK Scene is Complex

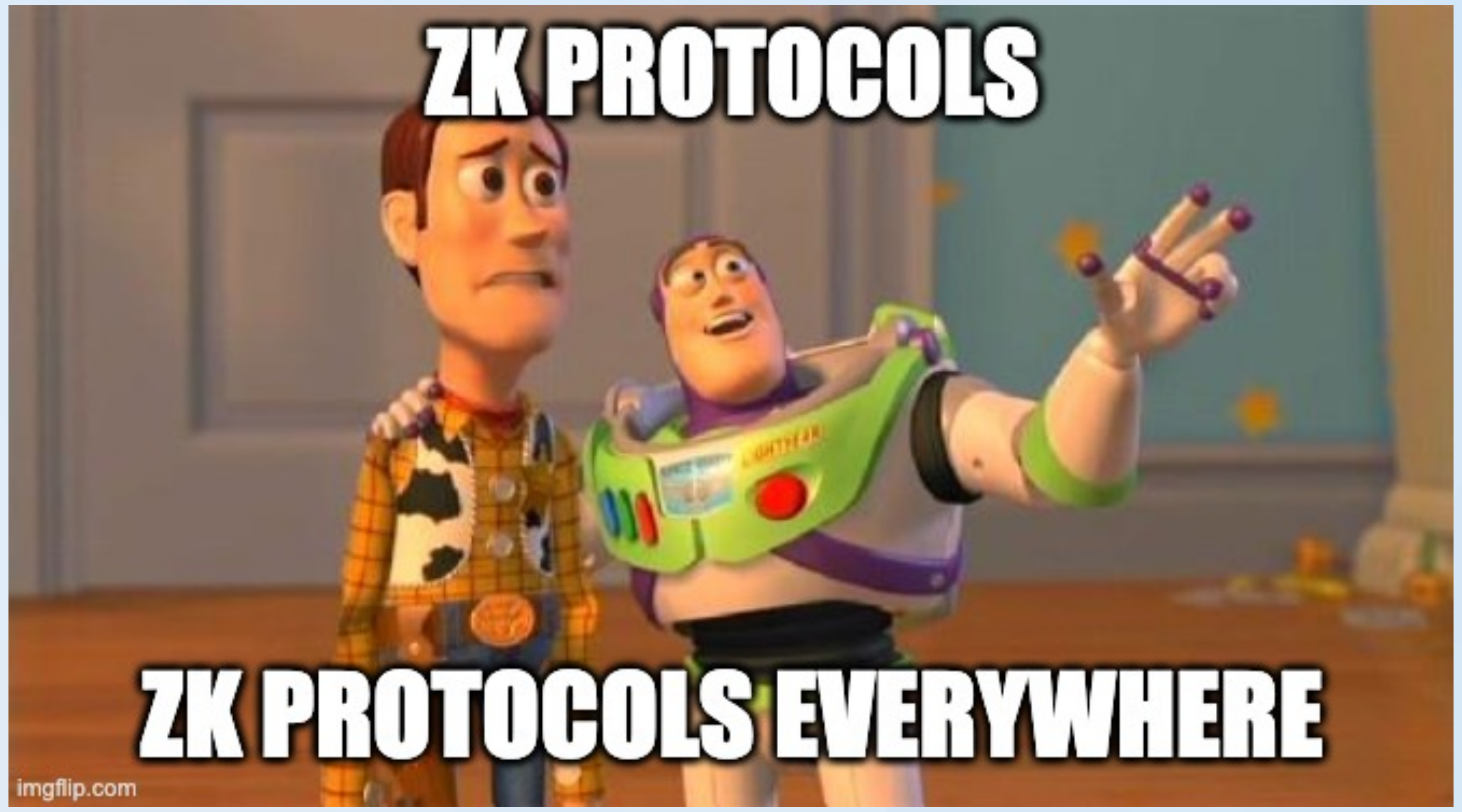
Several Categories of Protocols and Multiple Members per Category



Our Focus: zk-SNARKs

The ZK Scene is Complex

Several Categories of Protocols and Multiple Members per Category



Our Focus: zk-SNARKs

But what does this mean?

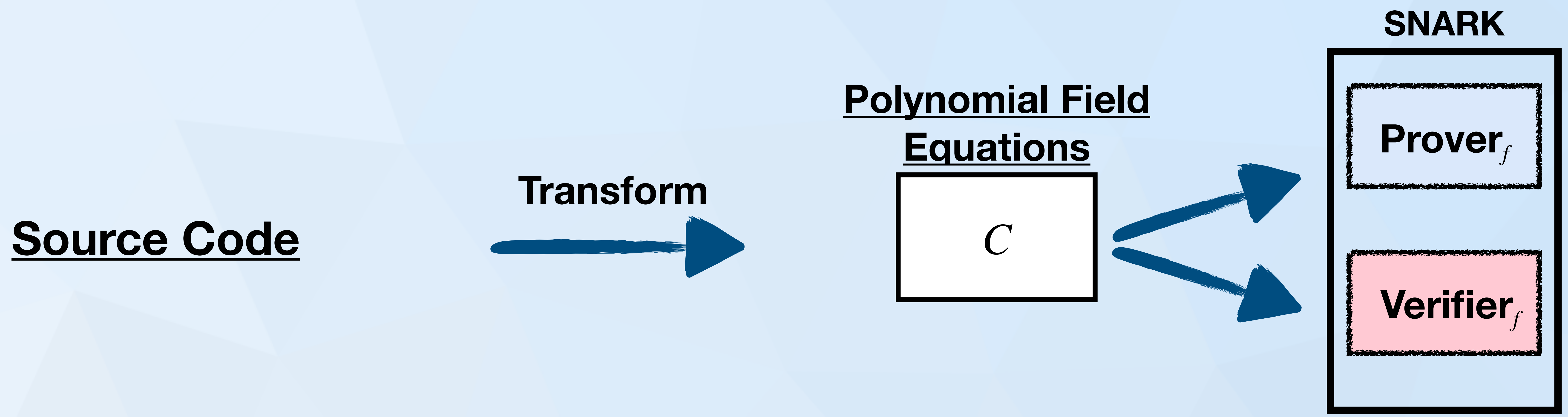
Why ZK Languages or Frameworks?

- **ZK proofs can significantly enhance Dapps**
  - The verifier can live on the blockchain while proofs can be submitted by anyone
  - But prover and verifier need to be customized on a Dapp basis
  - Need for creating custom "ZK protocols" without being a cryptography expert

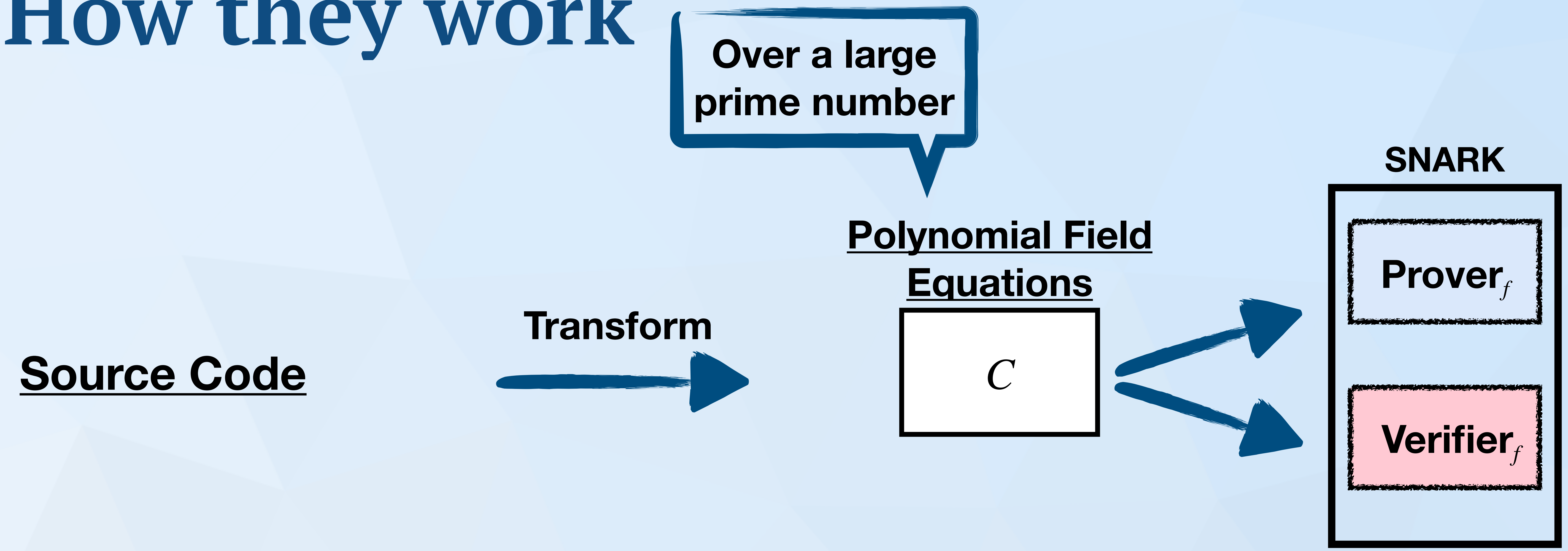
Privacy

Scalability

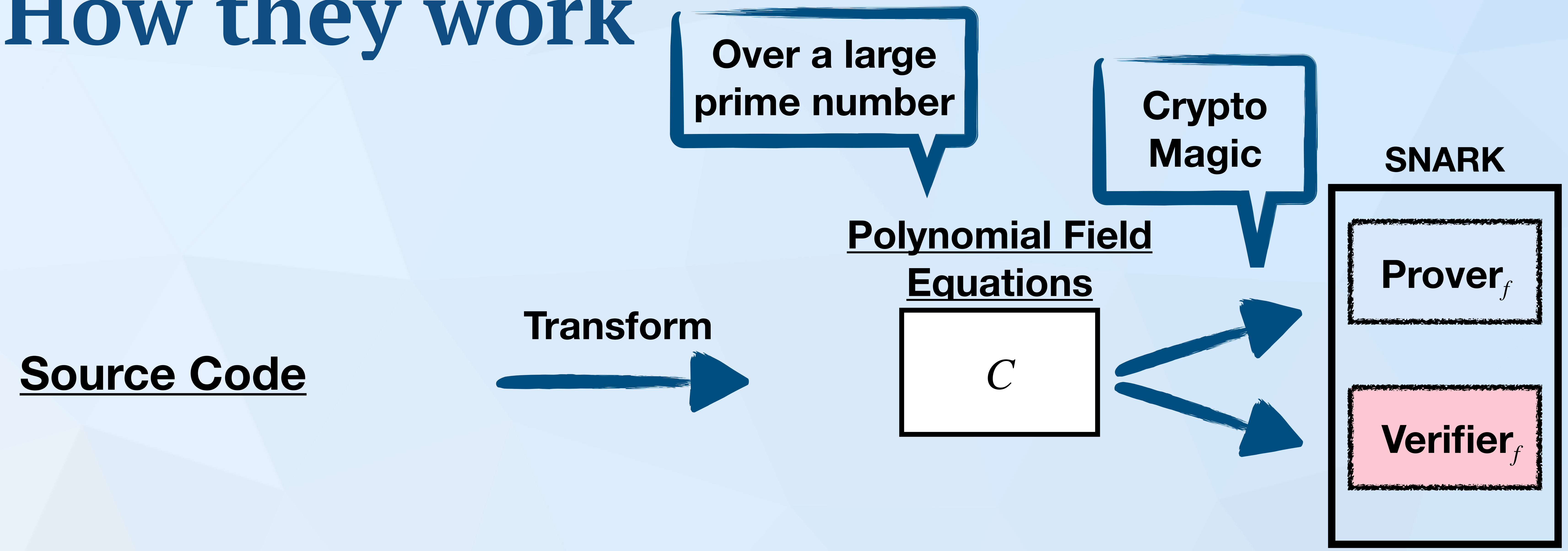
How they work



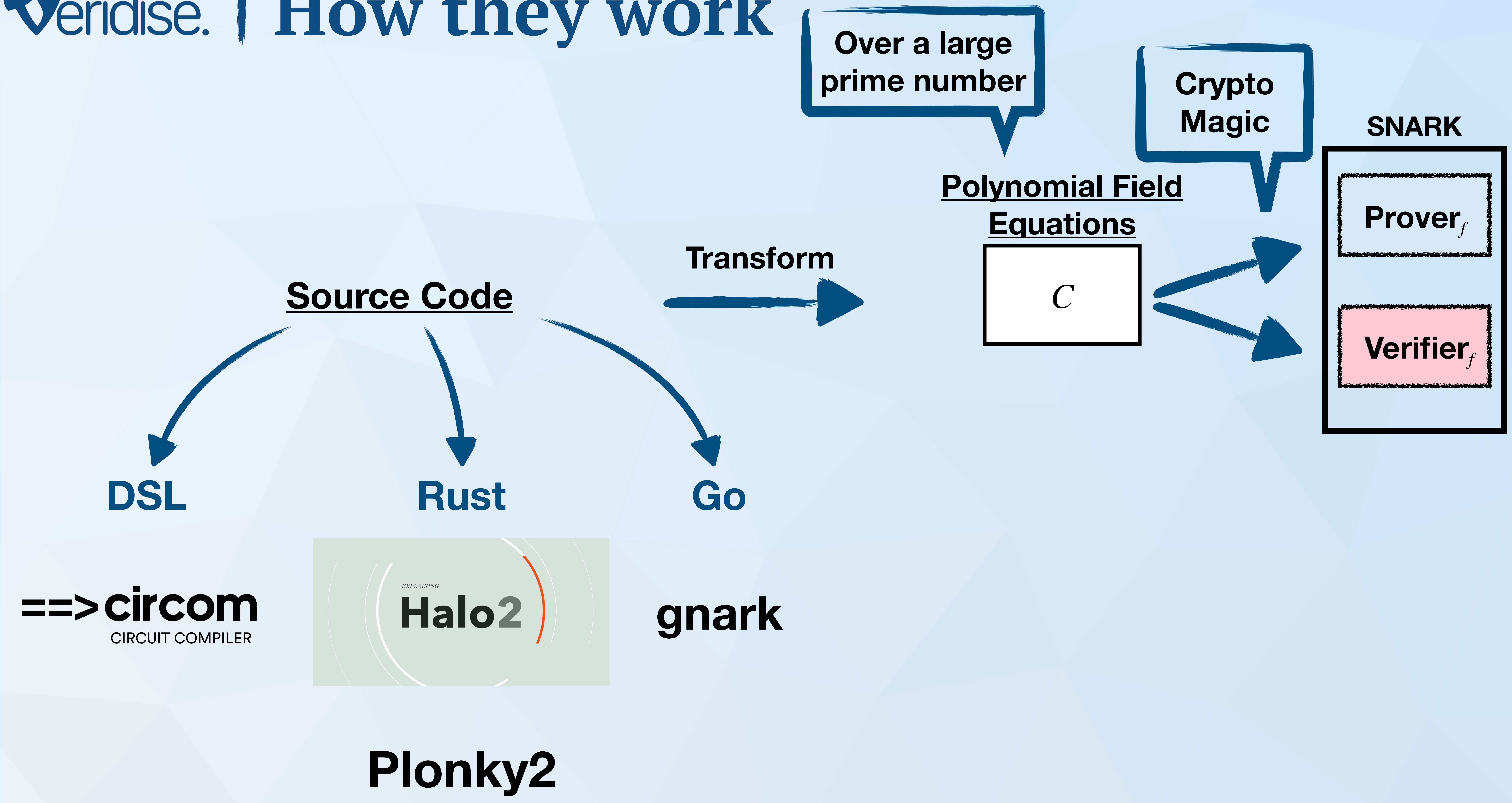
How they work



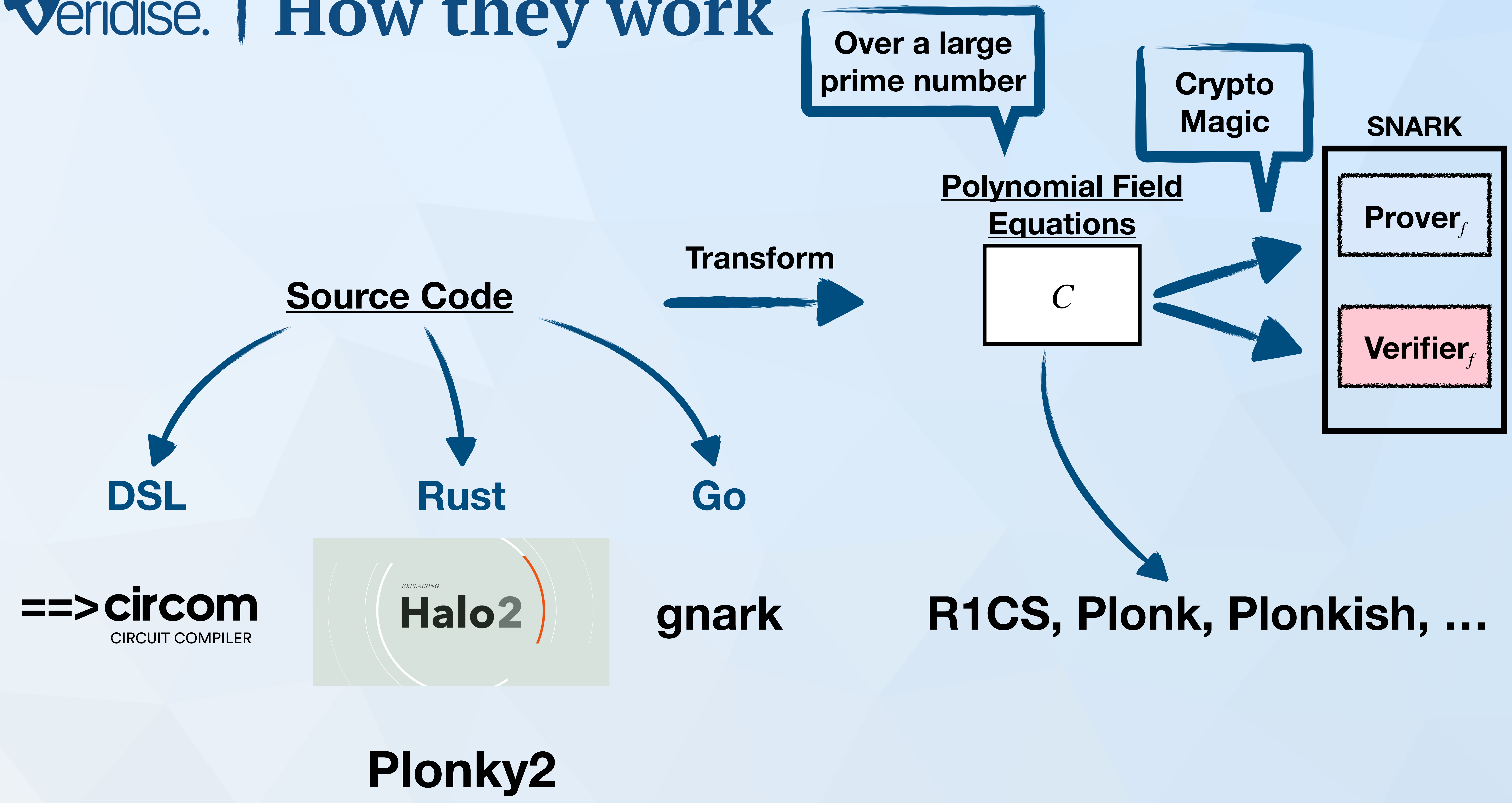
How they work



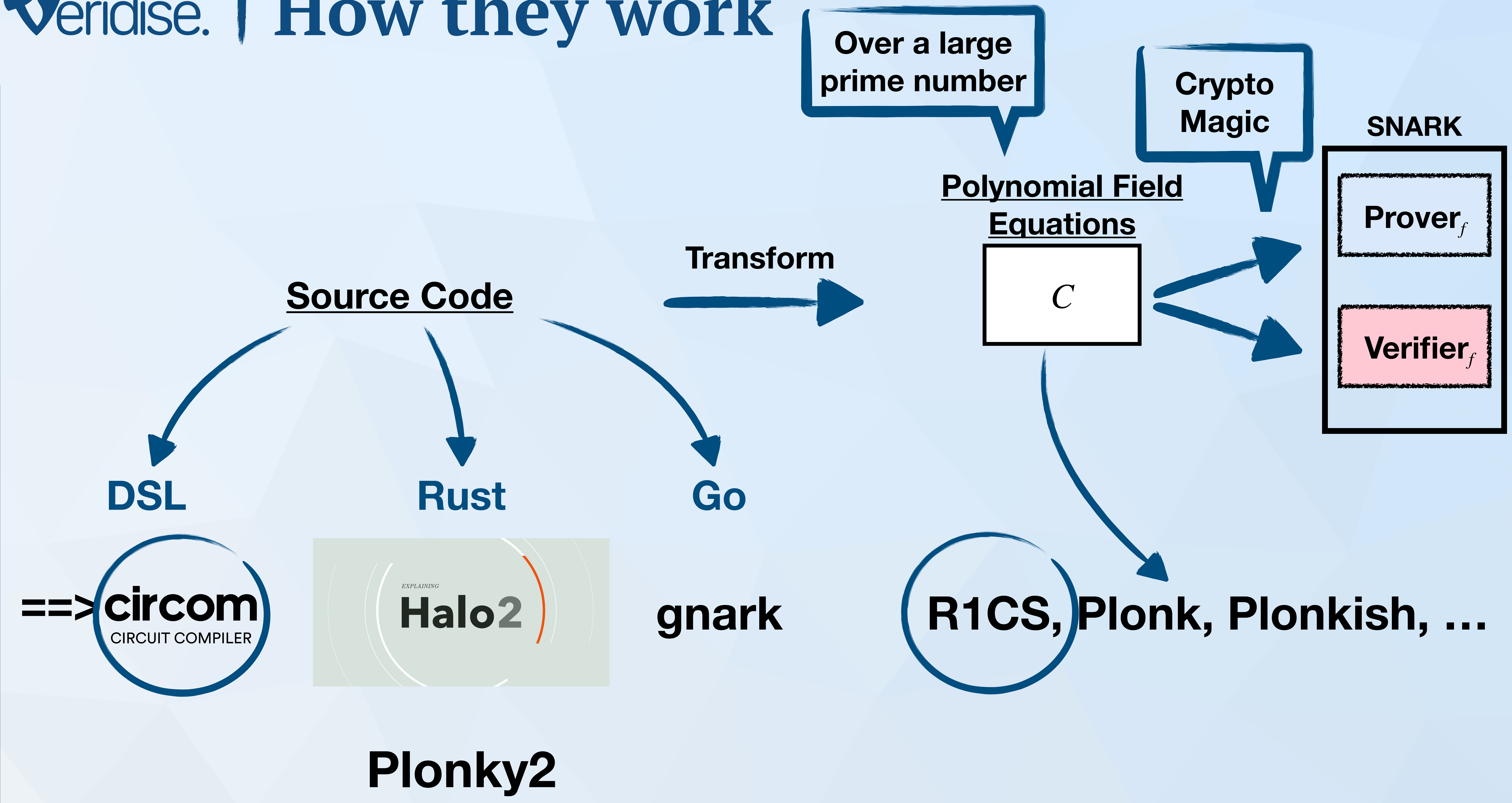
How they work



How they work



How they work



Circom 101

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

- Computation in circom is encoded as circuits
  - A circuit is a composition of templates
- Each template defines two things over signals
  - Constraints (checked by the verifier)
  - Witness generation (used to generate the proof)
- ***Attention:*** **ALL** operations are modulo a big prime. That is, a **op** b is really a **op** b % p
  - Therefore, all signals are between 0 and p-1

Veridise.

Zooming In The Circuit

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Zooming In The Circuit

Signals can be either input, output, or intermediate

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Zooming In The Circuit

Signals can be either input, output, or intermediate

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Output signals must have a witness assignment

Zooming In The Circuit

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Signals can be either input, output, or intermediate

Operator <-- only affects the witness generation

Output signals must have a witness assignment

Zooming In The Circuit

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Signals can be either input, output, or intermediate

Operator <-- only affects the witness generation

Operator === only affects the verifier

Output signals must have a witness assignment

Zooming In The Circuit

Signals can be either input, output, or intermediate

Operator <-- only affects the witness generation

Operator === only affects the verifier

```
pragma circom 2.0.0;

template Multiplier2 () {

  // Declaration of signals.
  signal input a;
  signal input b;
  signal tmp;
  signal output c;

  // Witness generation
  c <-- a * b;

 // Constraints.
 c === a * b;
 tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Output signals must have a witness assignment

- Constraints must be expressible as A*B - C = 0, where A,B,C are linear expressions (at most quadratic)
  - This limitation stems from R1CS

Veridise. |

Zooming In The Circuit

Signals can be either input, output, or intermediate

Operator <-- only affects the witness generation

Operator === only affects the verifier

Component holds a template instance

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

Output signals must have a witness assignment

- Constraints must be expressible as A*B - C = 0, where A,B,C are linear expressions (at most quadratic)
  - This limitation stems from R1CS

Veridise.

Zooming In The Circuit

Signals can be either input, output, or intermediate

Operator <-- only affects the witness generation

Operator === only affects the verifier

Component holds a template instance

```
pragma circom 2.0.0;

template Multiplier2 () {

  // Declaration of signals.
  signal input a;
  signal input b;
  signal tmp;
  signal output c;

  // Witness generation
  c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```

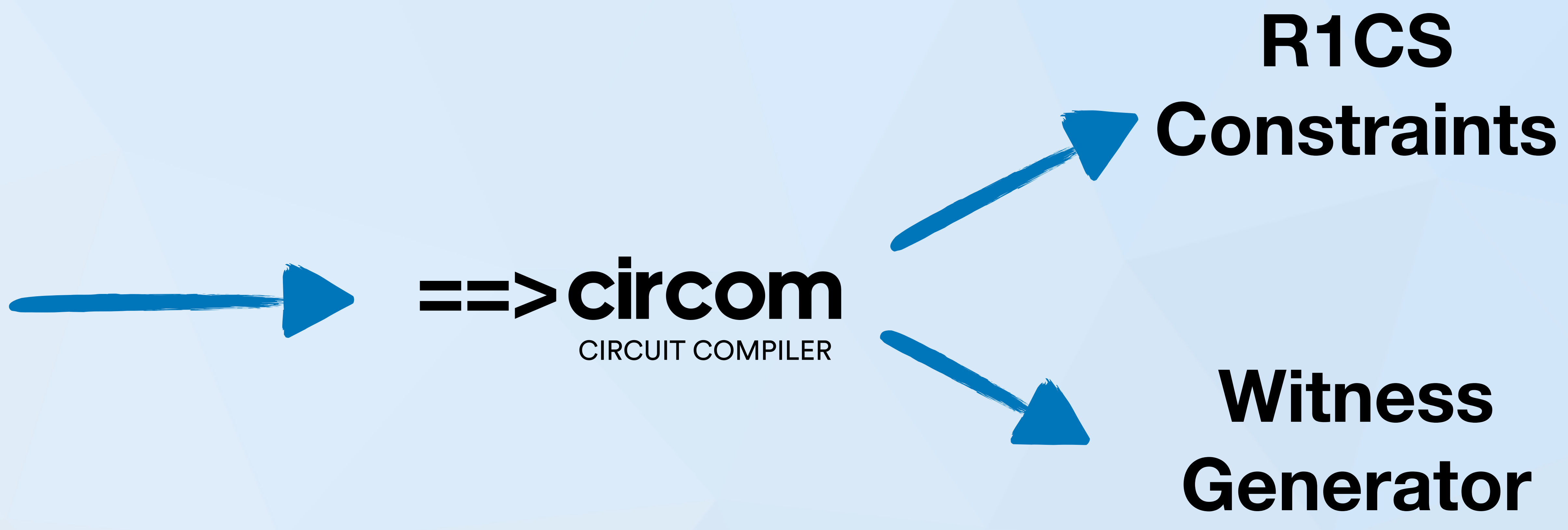
- Constraints must be expressible as A*B - C = 0, where A,B,C are linear expressions (at most quadratic)
  - This limitation stems from R1CS

Output signals must have a witness assignment

Signals not in this list are private

What to do next?

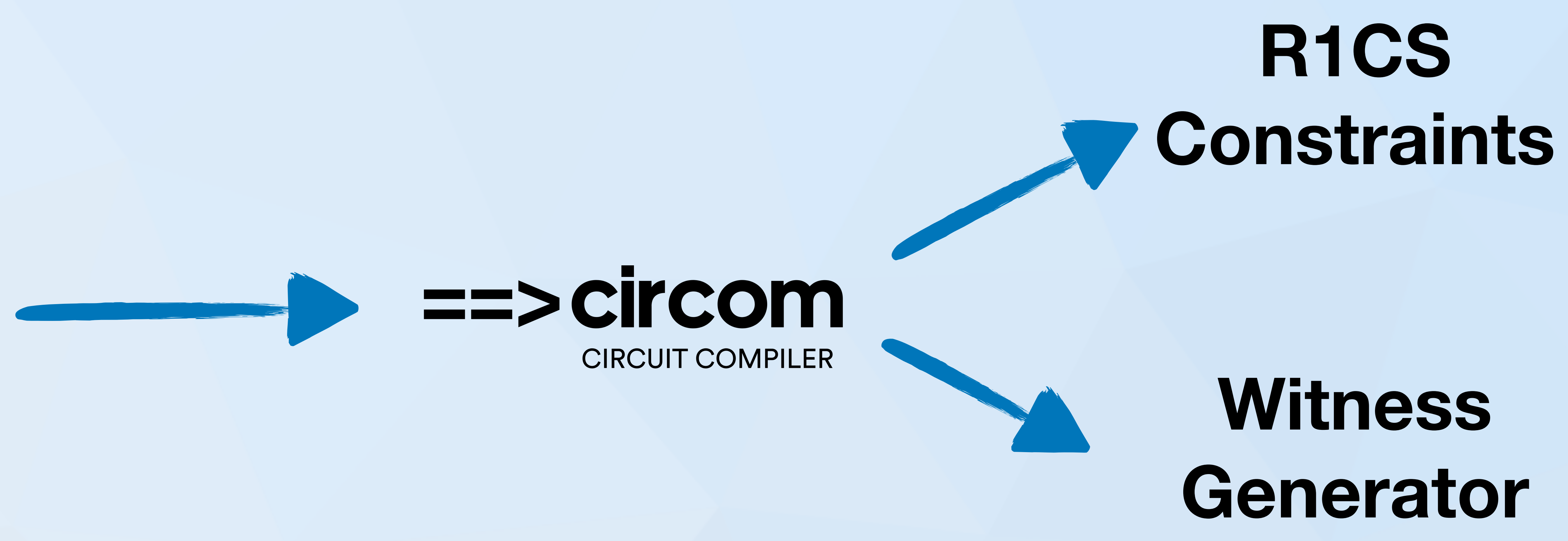
```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```



What to do next?

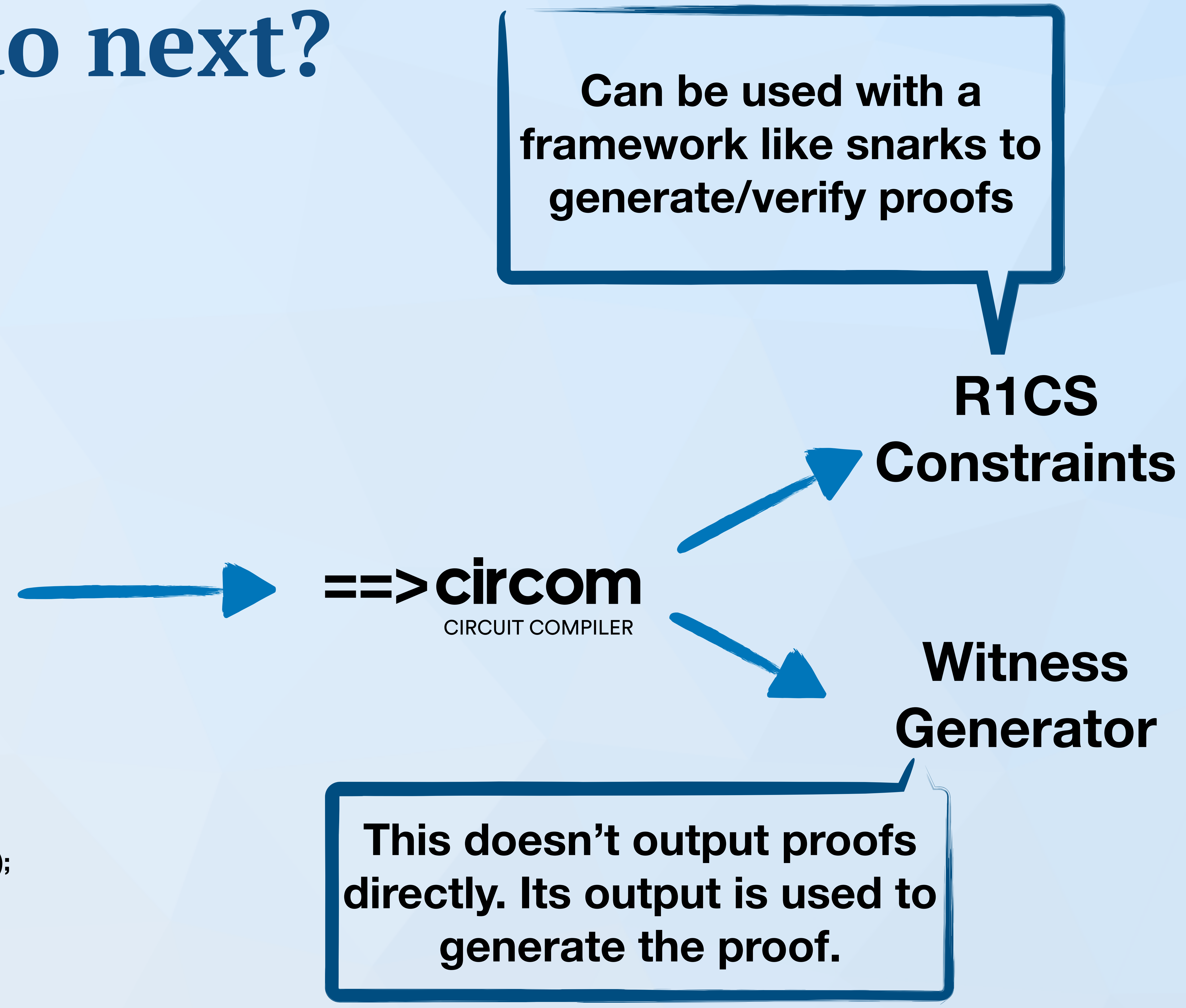
Can be used with a framework like snarks to generate/verify proofs

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```



What to do next?

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```



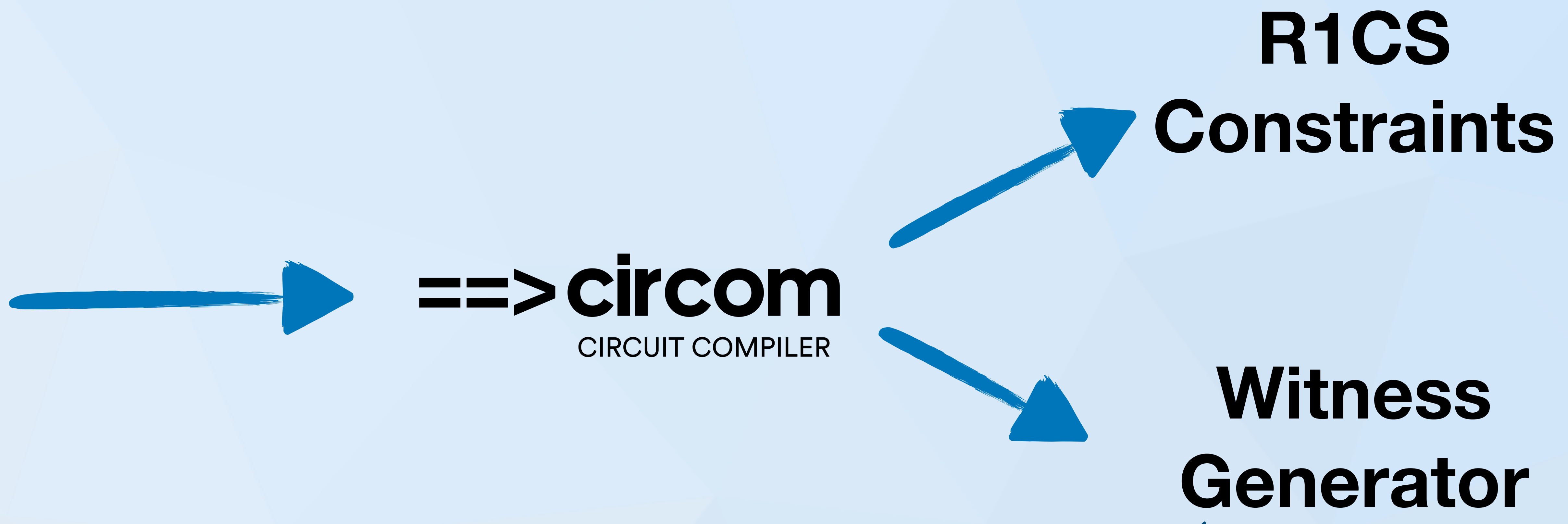
Can be used with a framework like snarks to generate/verify proofs

This doesn't output proofs directly. Its output is used to generate the proof.

What to do next?

Can be used with a framework like snarks to generate/verify proofs

```
pragma circom 2.0.0;

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal tmp;
   signal output c;

   // Witness generation
   c <-- a * b;

  // Constraints.
  c === a * b;
  tmp === 0;
}

component main {public [a]} = Multiplier2();
```



This doesn't output proofs directly. Its output is used to generate the proof.

Detailed Instructions

More circom features

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

We can define arrays of signals

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

We can also have local variables

We can define arrays of signals

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

Veridise.

More circom features

Templates can have parameters

We can also have local variables

And control-flow

We can define arrays of signals

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

Veridise.

More circom features

Templates can have parameters

We can also have local variables

And control-flow

We can define arrays of signals

Local variables can help us build complex expressions

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

We can also have local variables

And control-flow

This is equivalent to:
out <-- sum; out === sum;

We can define arrays of signals

Local variables can help us build complex expressions

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

We can also have local variables

And control-flow

This is equivalent to:
out <-- sum; out === sum;

We can define arrays of signals

Local variables can help us build complex expressions

sum is equiv to in[0] + in[1] + … in[n-1]

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

Templates can have parameters

We can define arrays of signals

We can also have local variables

Local variables can help us build complex expressions

And control-flow

This is equivalent to:
out <-- sum; out === sum;

sum is equiv to in[0] + in[1] + ... in[n-1]

We can also compose templates

```
template SumN(n) {
  signal input ins[n];
  signal output out;

  var sum = 0;

  for (var i = 0; i < n; i++) { sum += ins[i]; }

  out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

More circom features

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

Templates can have parameters

We can define arrays of signals

We can also have local variables

Local variables can help us build complex expressions

And control-flow

This is equivalent to:
out <-- sum; out === sum;

sum is equiv to in[0] + in[1] + … in[n-1]

We can also compose templates

We simply need to constraint/initialize all the inputs of the sub-component

Veridise.

More circom features

Templates can have parameters

We can define arrays of signals

We can also have local variables

Local variables can help us build complex expressions

And control-flow

This is equivalent to: out <-- sum; out === sum;

sum is equiv to in[0] + in[1] + … in[n-1]

We can also compose templates

Then, we can also constraint their output

We simply need to constraint/initialize all the inputs of the sub-component

```
template SumN(n) {
   signal input ins[n];
   signal output out;

   var sum = 0;

   for (var i = 0; i < n; i++) { sum += ins[i]; }

   out <== sum;
}

template Foo() {
  signal input ins[5];
  component sum5 = SumN(5);

  for (var i = 0; i < 5; i++) { sum5.ins[i] <== ins[i]; }
  sum5.out === 50;
}

component main = Foo();
```

Veridise.

Start Thinking in Circom

One of the most common mistakes is that people think in terms of traditional programming

```
void AssertBinary(int in) {

  assert(in == 0 || in == 1);

}
```

```
template AssertBinary {
  signal input in;



}
```

Start Thinking in Circom

One of the most common mistakes is that people think in terms of traditional programming

```
void AssertBinary(int in) {

    assert(in == 0 || in == 1);

}
```

```
template AssertBinary {
   signal input in;
   in * (1 - in) === 0;

}
```

Start Thinking in Circom

One of the most common mistakes is that people think in terms of traditional programming

```
void AssertBinary(int in) {

    assert(in == 0 || in == 1);

}
```

```
template AssertBinary {
   signal input in;
   in * (1 - in) === 0;

}
```

Can more things go wrong?
Well, of course :)
More on this later…

What's next?

- Even though circom seems like a small and simply language, we merely scratch the surface here.
- The best way to learn a language is to play with it!
  - We encourage you to do that by following the circom docs
    - Generate proofs, run the verifier, and generally poke around :)
  - If you are one of the RACE winners, you'll have to do so for the QUIZ
    - More info on the private discord channel
  - If you get stuck or have any question, just shoot a message on one of the discord channels (private or public)!