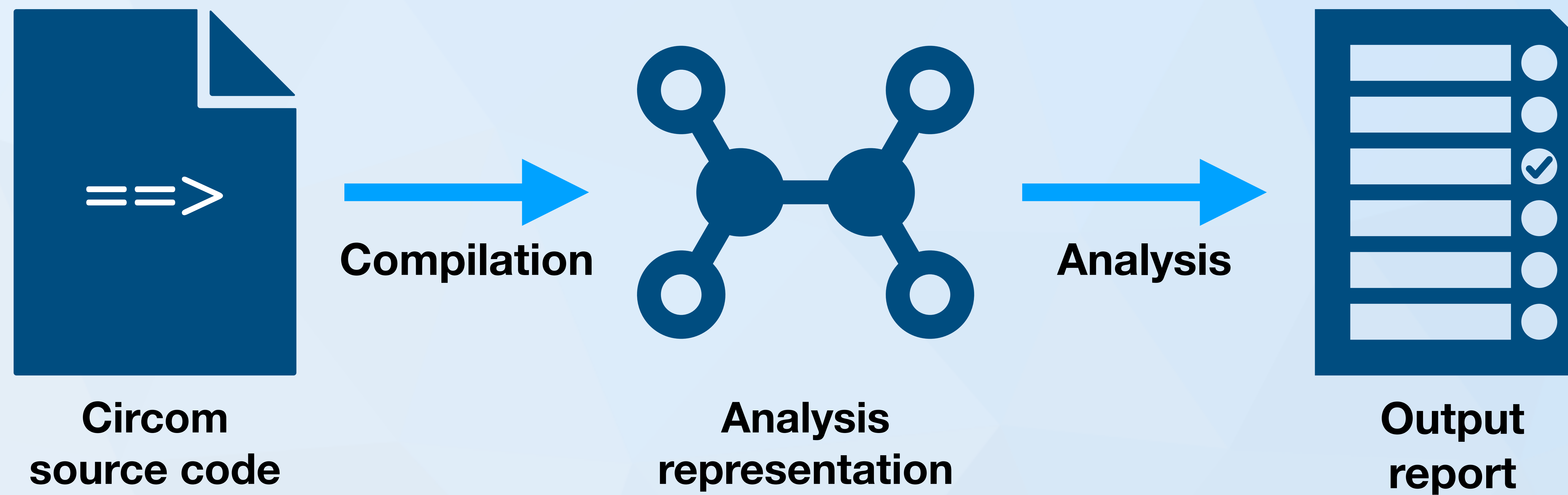
Static Analysis for ZK Circuits

Kostas Ferles
CRO, Veridise

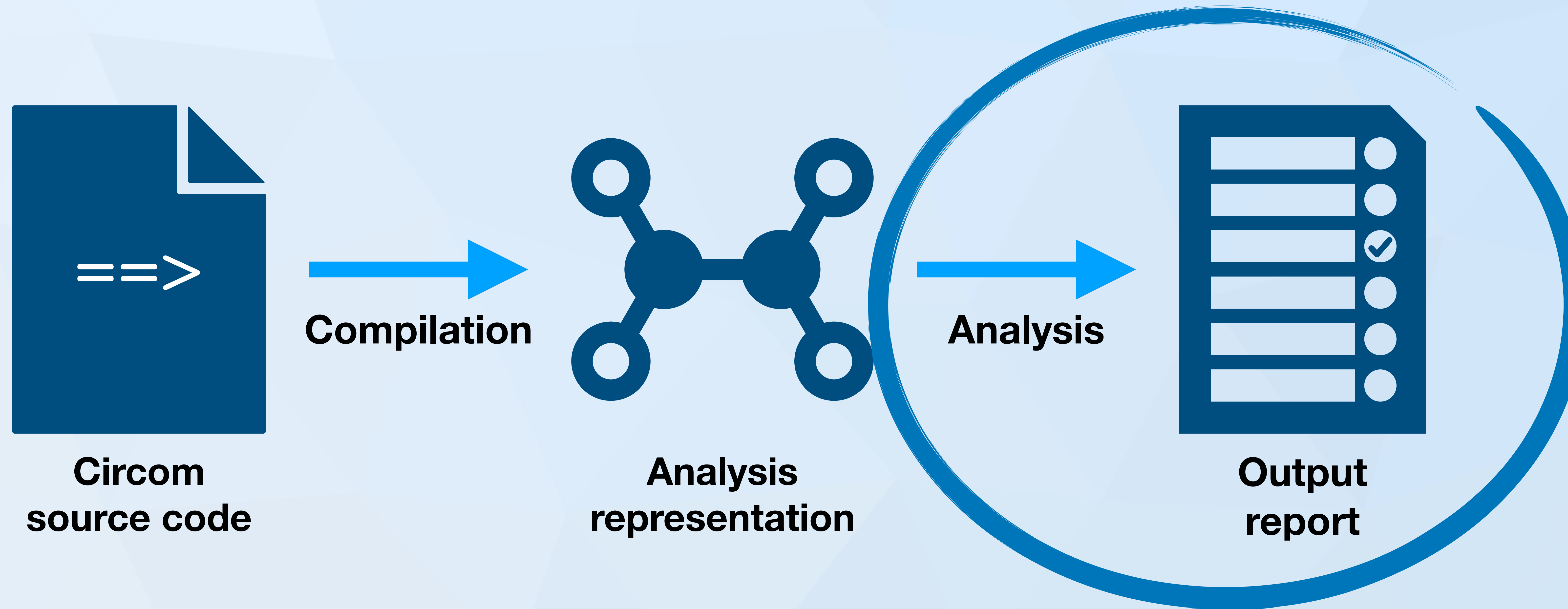
Veridise. | Recap

- You should have access to SaaS by now.
  - Let us know if there are any issues with this.
- Hope you had some fun staring at CDGs.
  - After this lecture, you will get access to more detectors.
  - No more staring at CDGs, let zkVanguard do the job for you.
  - Quiz 3 will be released after this lecture.

Veridise. | # Static Analysis Recap



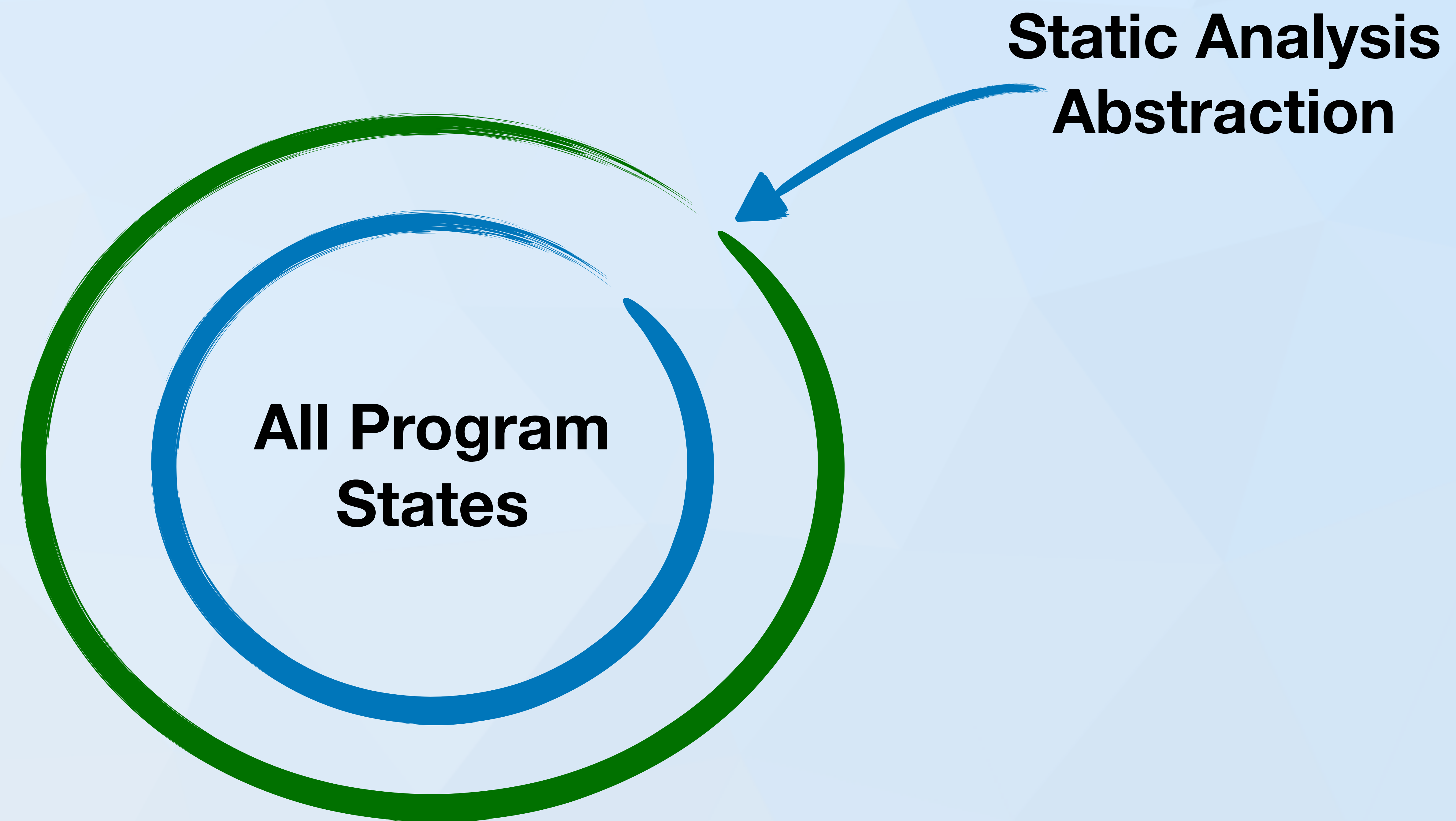
Veridise. | Static Analysis Recap



Veridise. |
Static Analysis Recap

All Program States

Veridise. | # Static Analysis Recap



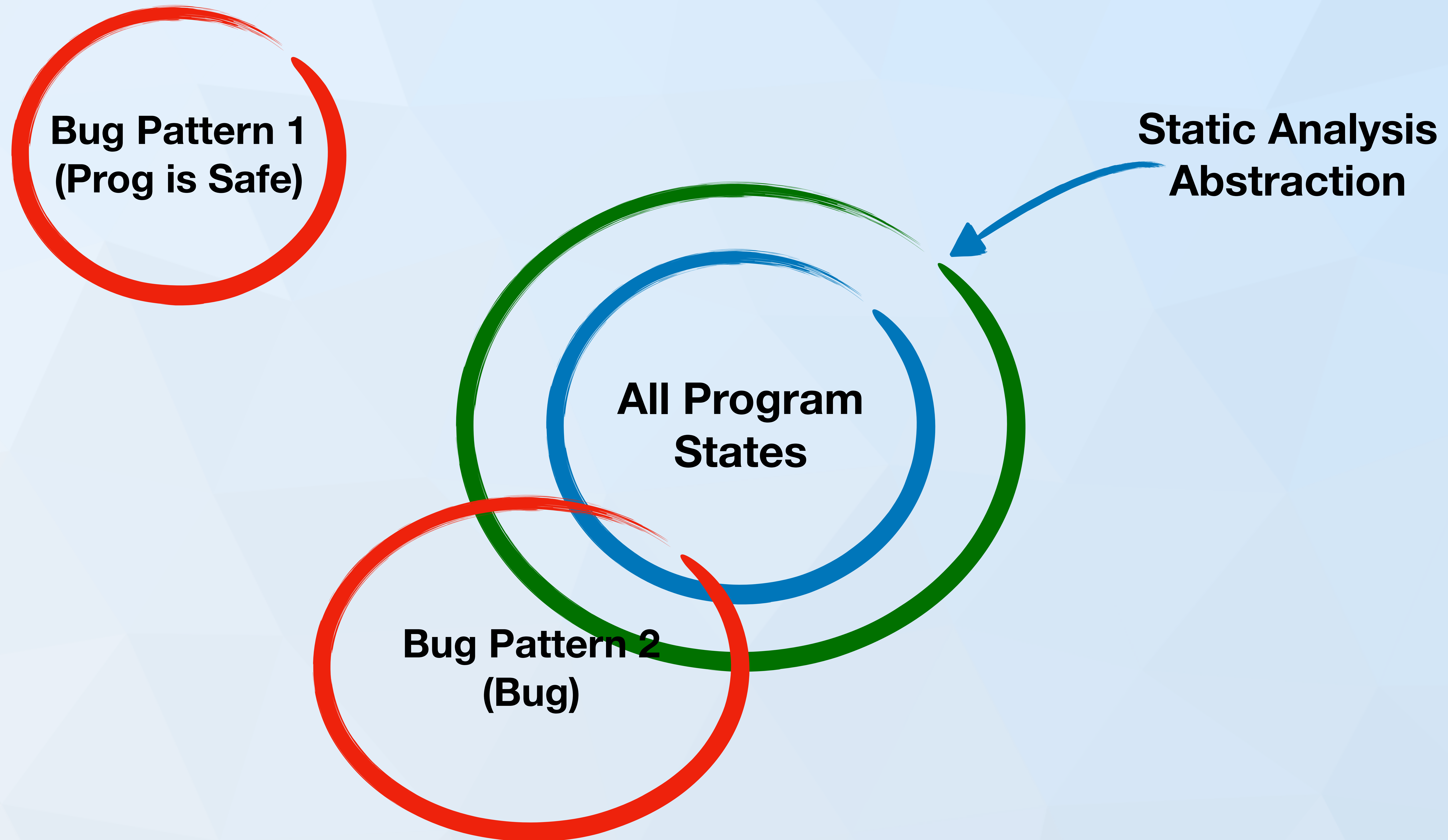
Veridise.

Static Analysis Recap

Bug Pattern 1
(Prog is Safe)

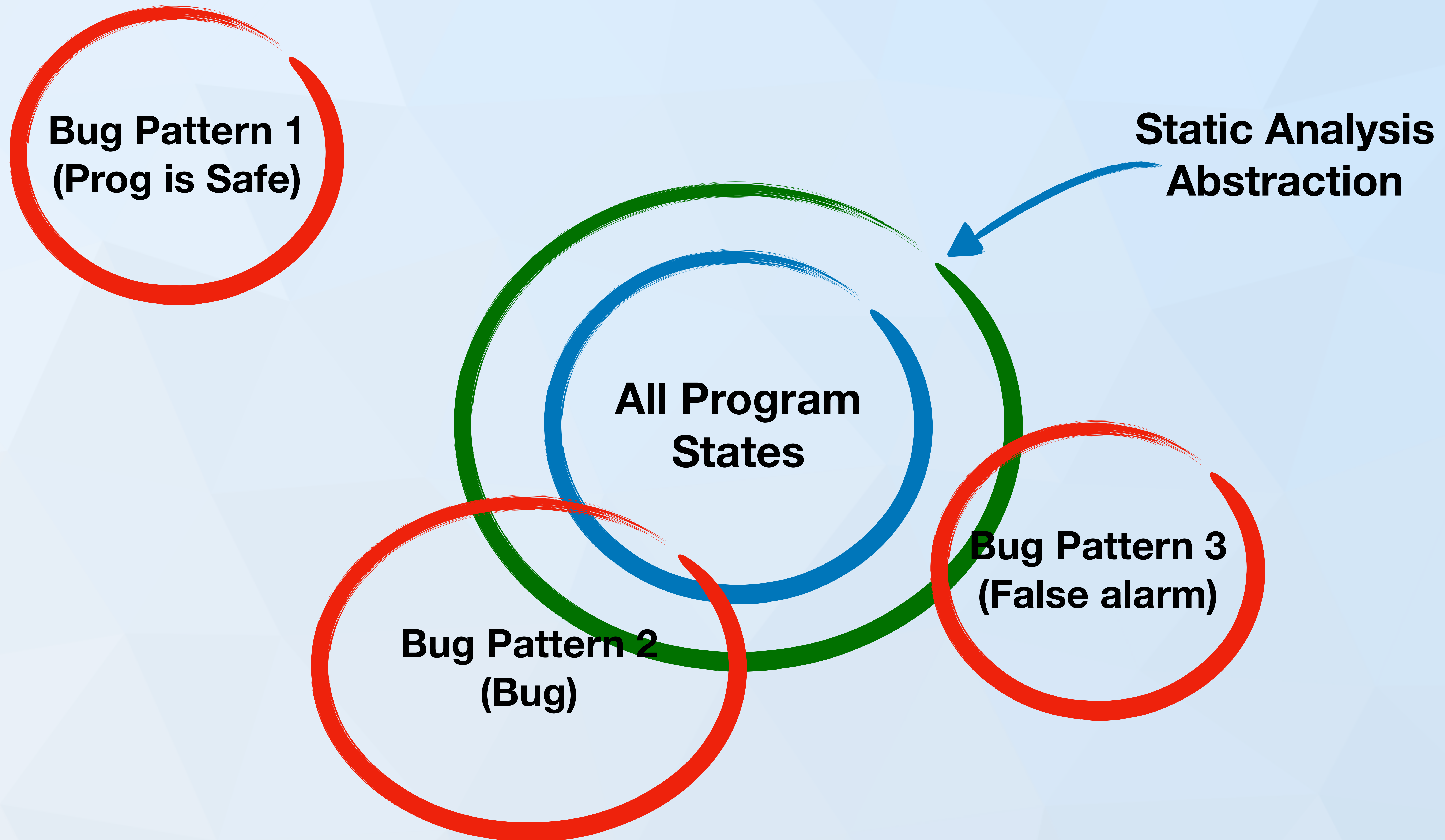
Static Analysis
Abstraction

Static Analysis Recap



Veridise.

Static Analysis Recap



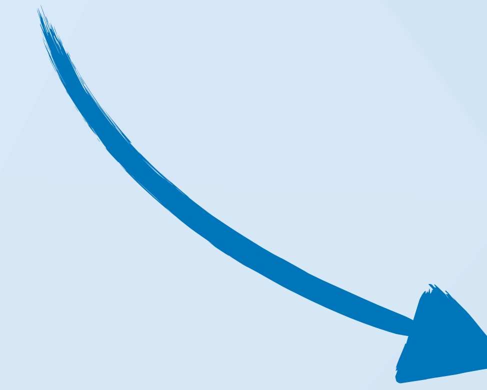
Veridise. | Let's Start With a Demo

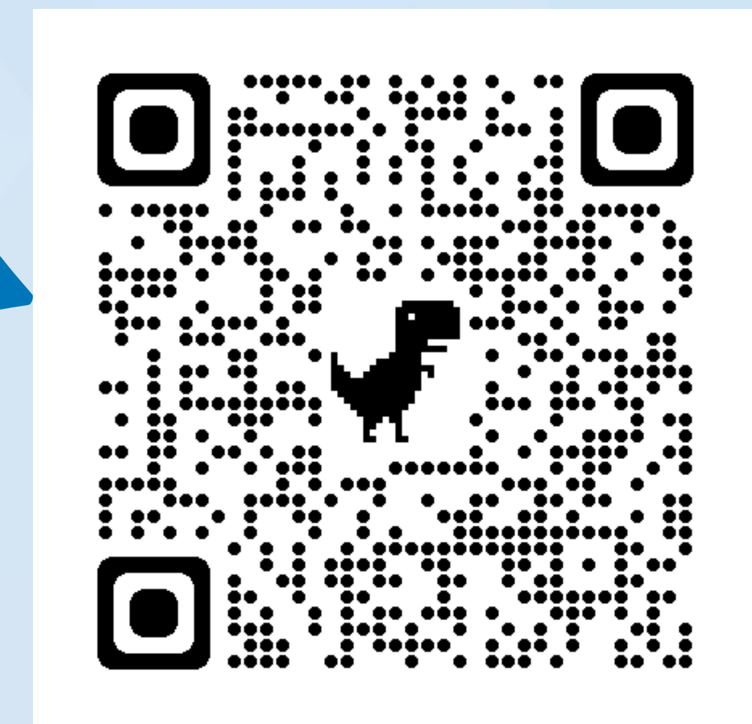
Doc example for under-constrained outputs

uco_example.circom

```
1  pragma circom 2.0.0;
2
3  template LowestBitIsOne() {
4    signal input inp;
5    signal output outp;
6
7    outp <-- inp & 1;
8    outp * (outp - 1) === 0;
9  }
10
11 component main = LowestBitIsOne();
```

Docs for all available to you!





Veridise. | # Quick Dive Into zk-Vanguard

- Most of zkVanguard detectors are simply looking for a path (or its absence) on a CDG.
  - For instance NDW, will look if a signal affects a branch.
- The logic of some detectors can be a bit more complex.
  - Let’s take a closer look to the uc-output detector.

Veridise. | The trivial case first

uco_example.circom

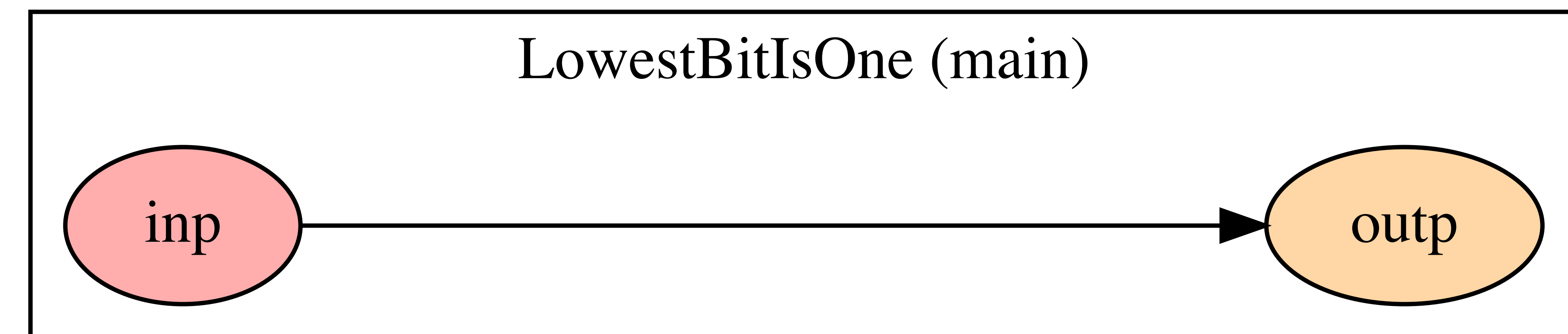
```
pragma circom 2.0.0;

template LowestBitIsOne() {
  signal input inp;
  signal output outp;

  outp <-- inp & 1;
  outp * (outp - 1) === 0;
}

component main = LowestBitIsOne();
```

The CDG!



- No constraint edge between inp and outp
- zkVanguard will report this as a bug

Veridise. | # Let's Complicate Things

```
1 pragma circom 2.0.0;
2
3 template Bar() {
4   signal input in;
5   signal output out1;
6   signal output out2;
7
8   out1 <== in + 5;
9   out2 <-- in & 1;
10  out2 * (out2 - 1) === 0;
11 }
12
13 template Foo() {
14   signal input inp;
15   signal output fout1;
16   signal output fout2;
17
18   component b = Bar();
19
20   b.in <== inp;
21
22   fout1 <== b.out1;
23   fout2 <== b.out2;
24 }
25
26 component main = Foo();
```

Let’s Complicate Things

out1 is fine!
out2 not so much

```
1 pragma circom 2.0.0;
2
3 template Bar() {
4   signal input in;
5   signal output out1;
6   signal output out2;
7
8   out1 <== in + 5;
9   out2 <-- in & 1;
10  out2 * (out2 - 1) === 0;
11 }
12
13 template Foo() {
14   signal input inp;
15   signal output fout1;
16   signal output fout2;
17
18   component b = Bar();
19
20   b.in <== inp;
21
22   fout1 <== b.out1;
23   fout2 <== b.out2;
24 }
25
26 component main = Foo();
```

Veridise. | Let's Complicate Things

out1 is fine!
out2 not so much

Now Foo is using Bar!
What should we do?

```
1 pragma circom 2.0.0;
2
3 template Bar() {
4   signal input in;
5   signal output out1;
6   signal output out2;
7
8   out1 <== in + 5;
9   out2 <-- in & 1;
10   out2 * (out2 - 1) === 0;
11 }
12
13 template Foo() {
14   signal input inp;
15   signal output fout1;
16   signal output fout2;
17
18   component b = Bar();
19
20   b.in <== inp;
21
22   fout1 <== b.out1;
23   fout2 <== b.out2;
24 }
25
26 component main = Foo();
```

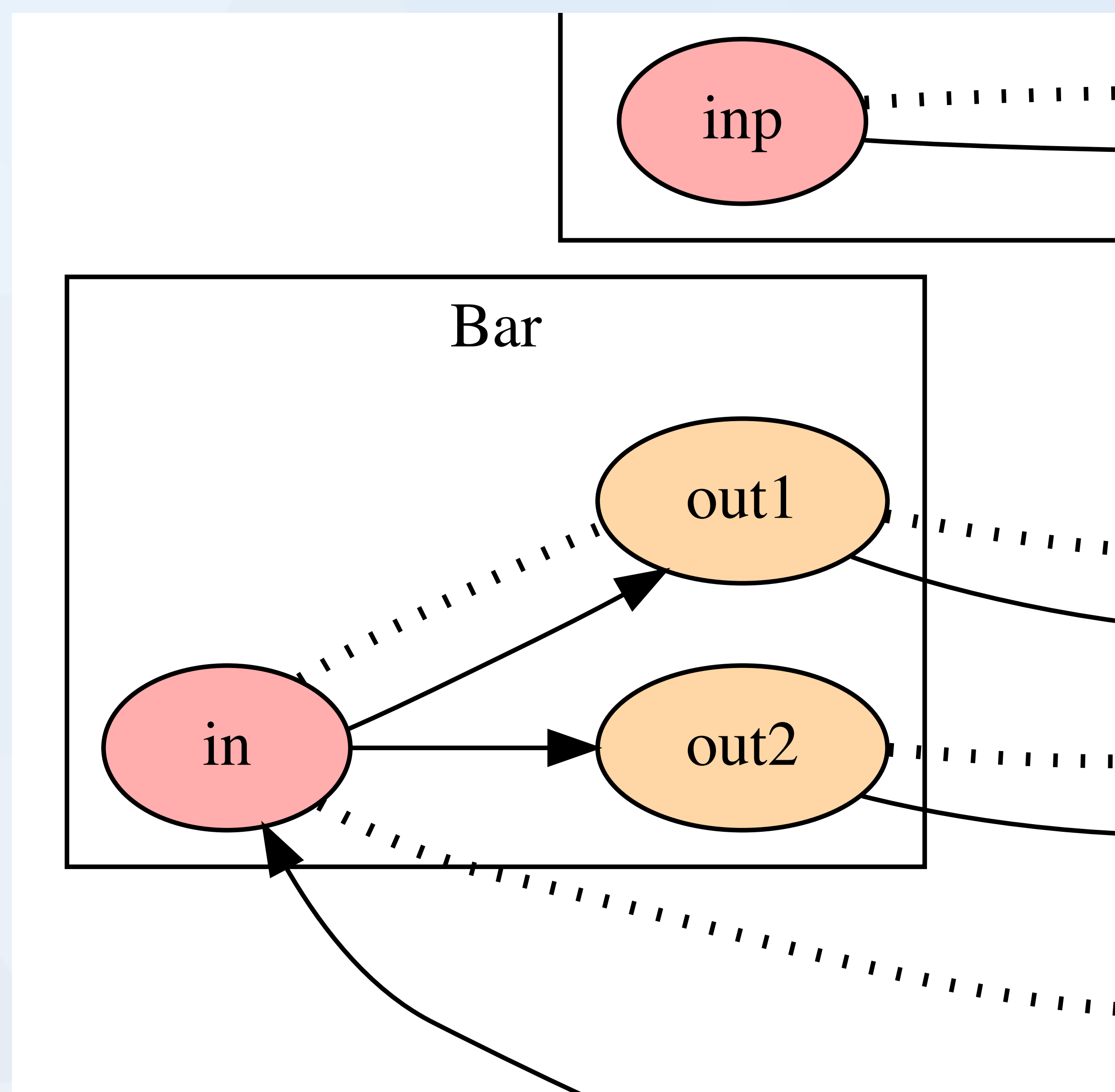
Veridise.

What should we do?

- Naive solution:
  - Every time you analyze a template, also analyze all its sub-components.
  - ***Problem***: This doesn't scale for real-world circuits. (Redundant computation)
- Can we do something smarter?
  - Well, glad you asked :)

The Smart(er) Solution

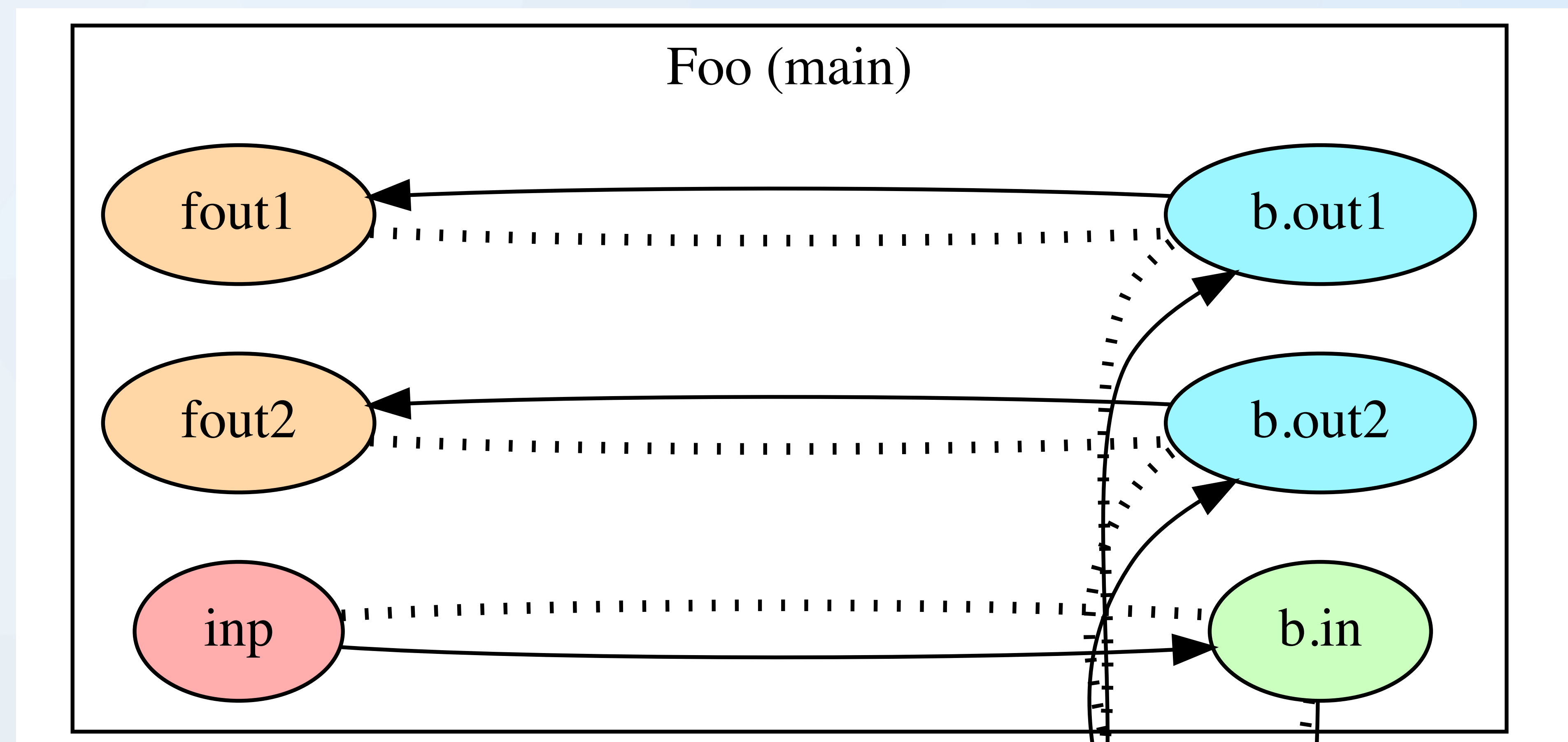
- Before you analyze a template ***T***:
  - Make sure you have analyzed all of its sub-components.
  - Every time you analyze a template, create a ***summary*** for its outputs.



- Summary for Bar:
  - out1 -> {in}, out2 -> {}
  - I.e., out1 is fine, out2 is not!

Veridise. | The Smart(er) Solution

- Now when you analyze ***T***:
  - No need to re-analyze its sub-components.
  - Just use their summary to propagate information for signals of ***T***



- Summary for Bar: out1 -> {in}, out2 -> {}
- Translate those signals in terms of Foo:
  - out1 is b.out1, out2 is b.out2, and in is b.in
- Therefor we can infer the following for Foo:
  - fout1 -> {inp}, fout2 -> {}
  - Note constraints involving inp, fout1, fout2 with signals from Bar

Veridise. | Let's see if zk-Vanguard agrees

```
----Preprocessing sources----
Running circom...
Done running circom
----Running Vanguard with dump-cdg,uc-outputs detector----
Running detector: dump-cdg
===================================================
 Vanguard's CDG Generator did not find any issues.
===================================================

Running detector: uc-outputs
=============================================================================
 Vanguard's unconstrained output signal detector found the following issues:
=============================================================================
[CRITICAL] In template Bar in foo.circom:3, Vanguard found an output signal that is unconstrained:
  * Signal out2

[CRITICAL] In template Foo in foo.circom:13, Vanguard found an output signal that is unconstrained:
  * Signal out2
```

Veridise. |

That's all!

- Let us know if you have any question!
- For the RACE winners:
  - Hope you have fun navigating some more complex code-bases with zk-Vanguard
  - And it's time to also write some circom on your own ;)