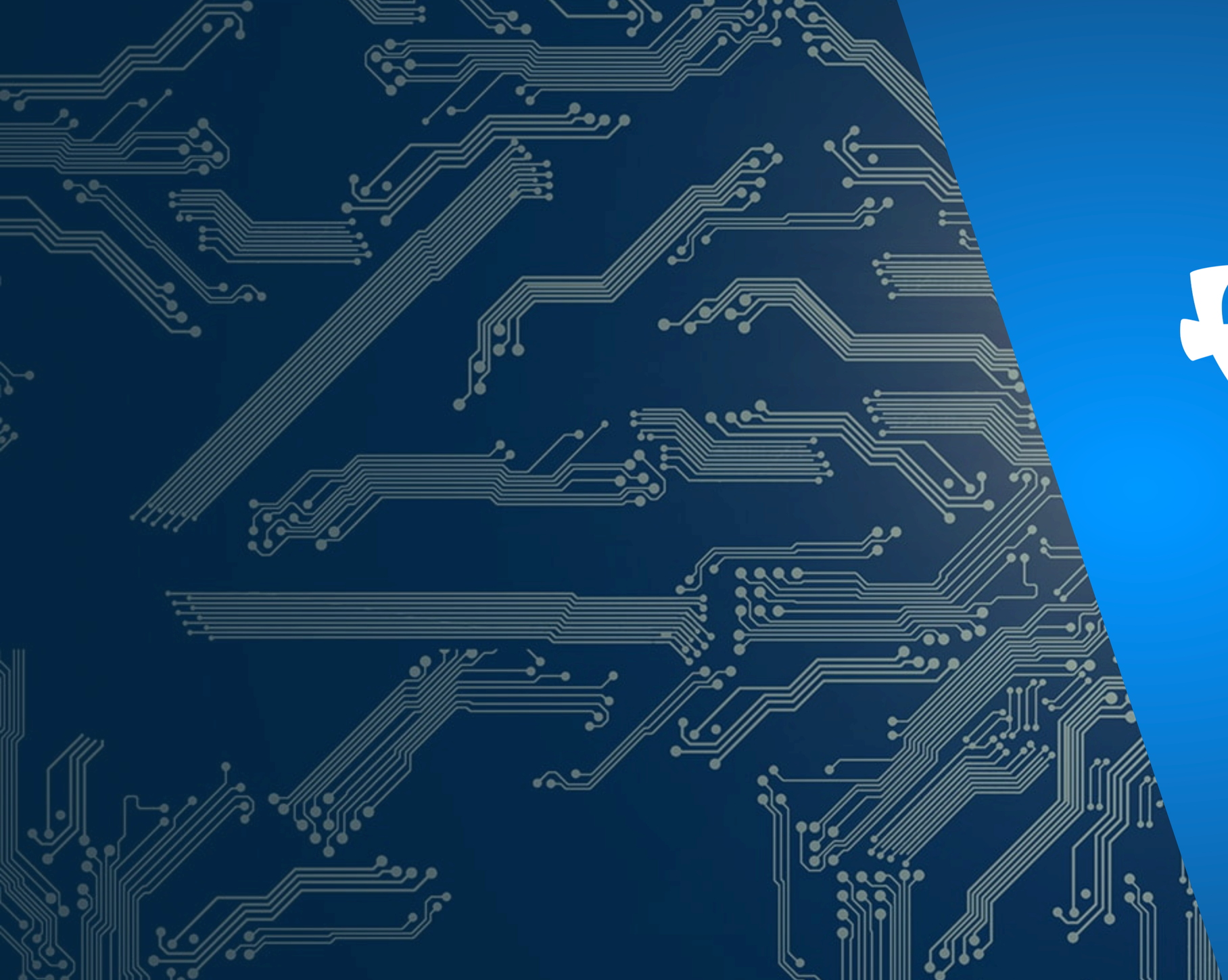


Veridise.

Circom Security

Jon Stephens
CTO, Veridise

Veridise. | Why should I care?

Tornado Cash

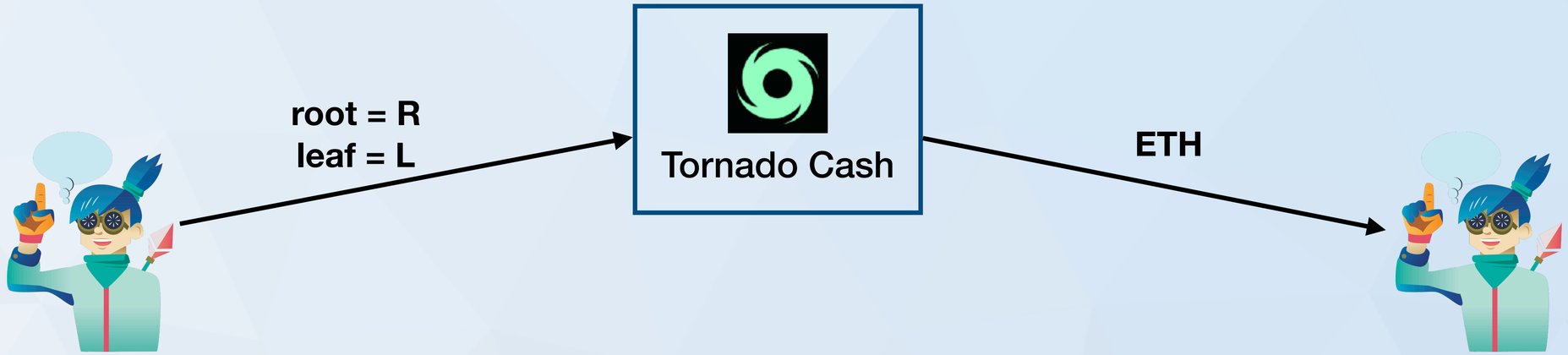
Oct 12, 2019 · 3 min read · Listen

Tornado.cash got hacked. By us.

Veridise. | Why should I care?

2 circuits/mimcsponge.circom

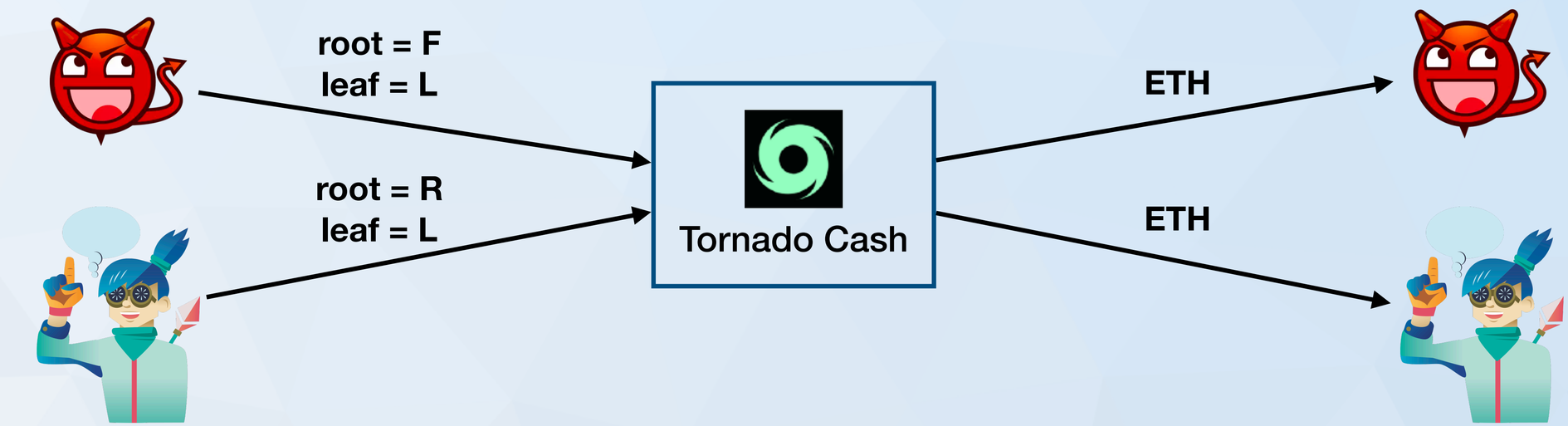
```
@@ -21,7 +21,7 @@ template MiMCSponge(nInputs, nRounds, nOutputs) {
21        }                                          21        }
22      }                                            22      }
23                                                   23
24  -   outs[0] = S[nInputs - 1].xL_out;            24  +   outs[0] <== S[nInputs - 1].xL_out;
25                                                   25
26      for (var i = 0; i < nOutputs - 1; i++) {     26      for (var i = 0; i < nOutputs - 1; i++) {
27        S[nInputs + i] = MiMCFeistel(nRounds);     27        S[nInputs + i] = MiMCFeistel(nRounds);
```



Veridise. | Why should I care?

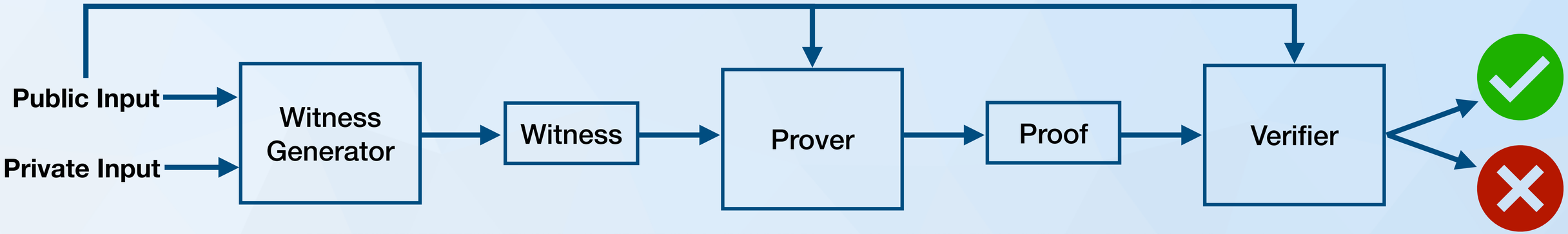
2 circuits/mimcsponge.circom

```
@@ -21,7 +21,7 @@ template MiMCSponge(nInputs, nRounds, nOutputs) {
21        }                                          21        }
22      }                                            22      }
23                                                   23
24  -   outs[0] = S[nInputs - 1].xL_out;             24  +   outs[0] <== S[nInputs - 1].xL_out;
25                                                   25
26      for (var i = 0; i < nOutputs - 1; i++) {     26      for (var i = 0; i < nOutputs - 1; i++) {
27        S[nInputs + i] = MiMCFeistel(nRounds);     27        S[nInputs + i] = MiMCFeistel(nRounds);
```

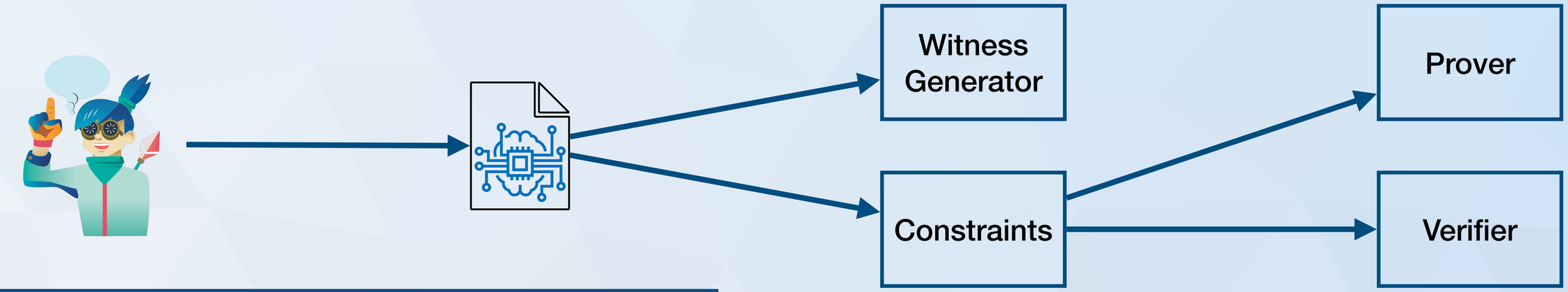


Veridise. | # ZK Circuits

ZK Circuit Usage

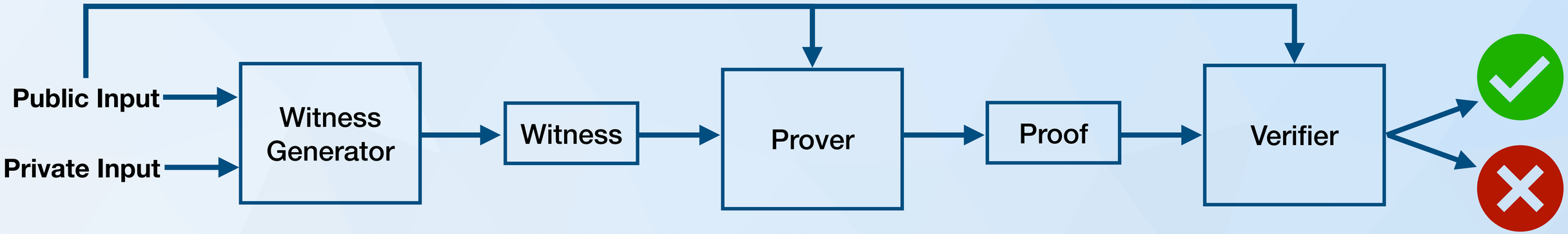


Creating a ZK Circuit



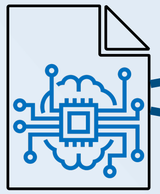
Veridise. | ZK Circuits

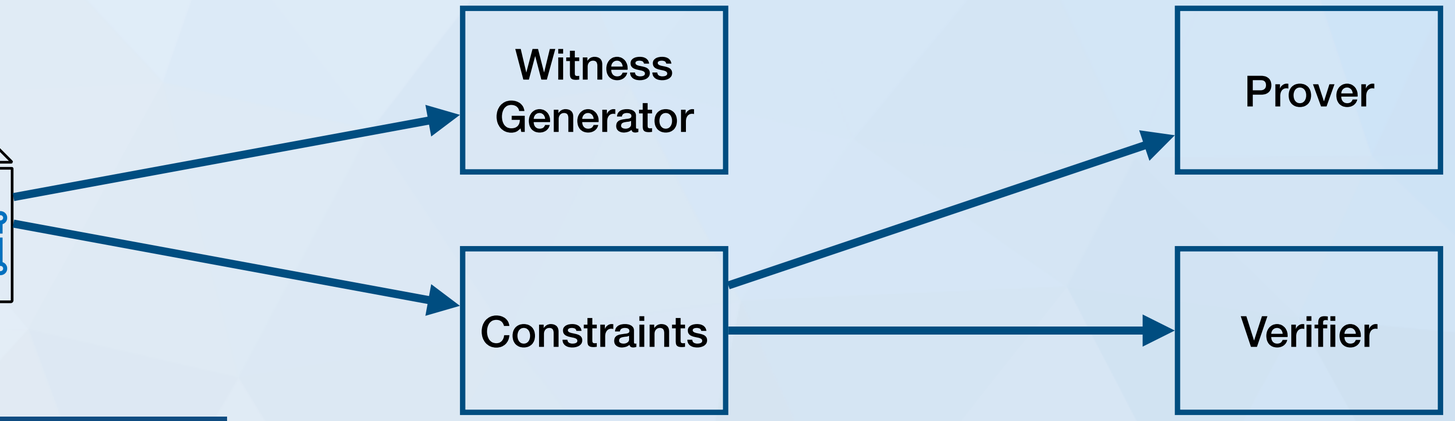
ZK Circuit Usage



Functional Correctness Violation

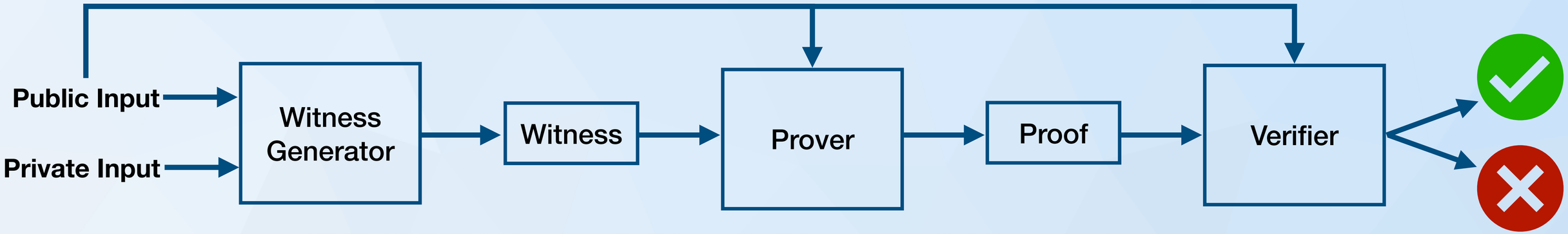
≠



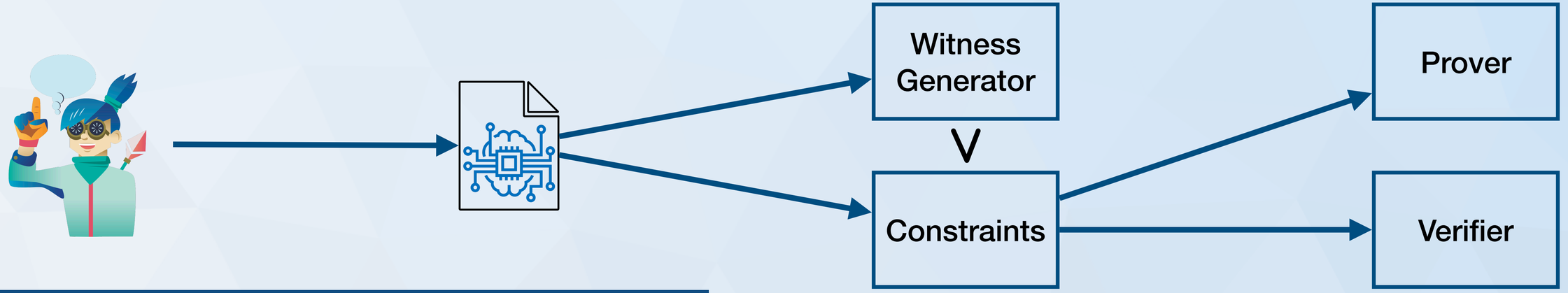


Veridise. | ZK Circuits

ZK Circuit Usage

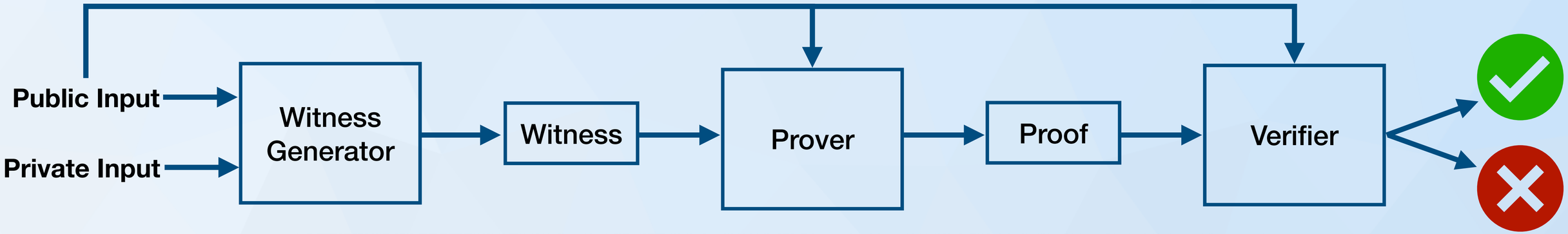


Over-constrained Circuit

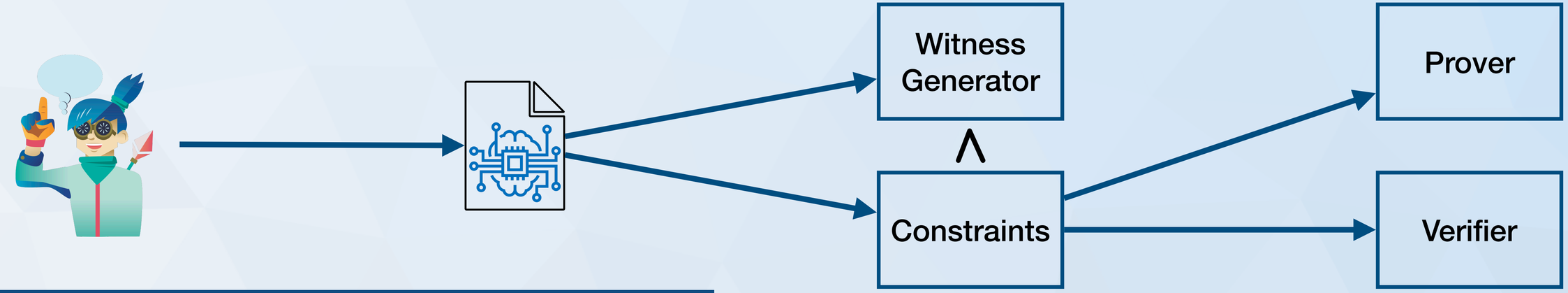


Veridise. | ZK Circuits

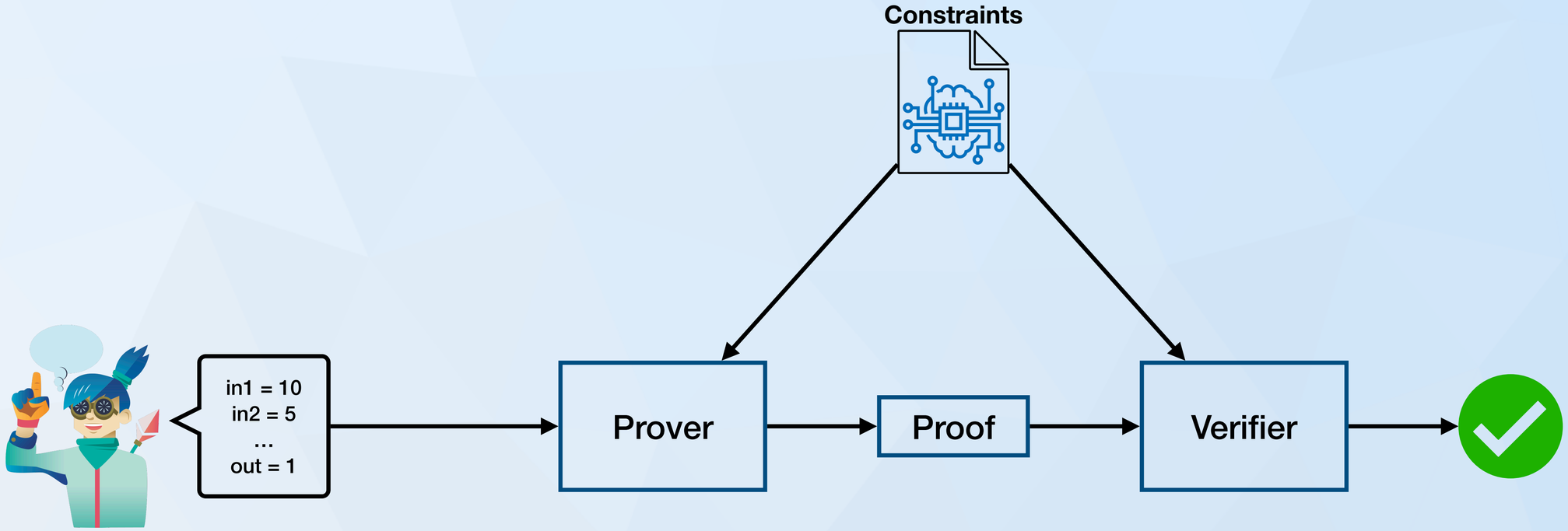
ZK Circuit Usage



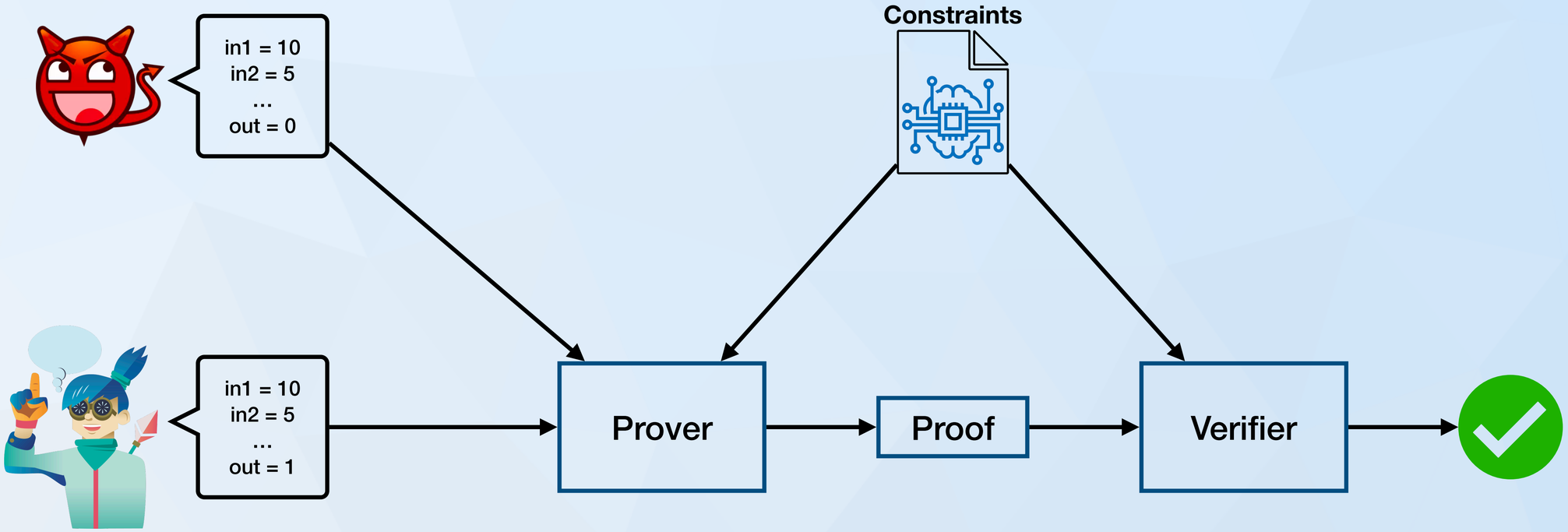
Under-constrained Circuit



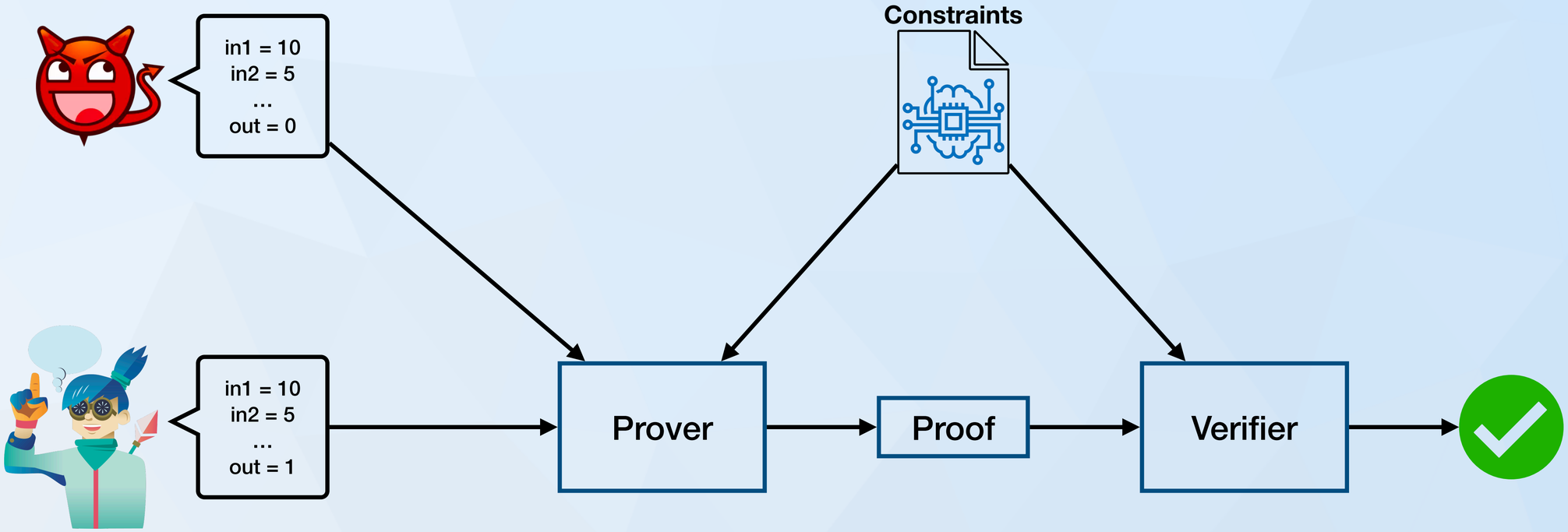
Veridise. | Underconstrained Circuits



Underconstrained Circuits



Underconstrained Circuits



Sometimes such circuits are desired, but must be very careful!

Veridise. | Unconstrained Output

```
template ArrayAdd(n) {
    signal input a[n];
    signal input b[n];
    signal output out[n];

    for (var i = 0; i < n - 1; i++) {
        out[i] <== a[i] + b[i];
    }
}
```

Unconstrained Output

```
template ArrayAdd(n) {
    signal input a[n];
    signal input b[n];
    signal output out[n];

    for (var i = 0; i < n - 1; i++) {
        out[i] <== a[i] + b[i];
    }
}
```

ArrayAdd(3) witness:

a = {1, 2, 3}

b = {6, 5, 4}

out = {7, 7, 7}

Unconstrained Output

```
template ArrayAdd(n) {
    signal input a[n];
    signal input b[n];
    signal output out[n];

    for (var i = 0; i < n - 1; i++) {
        out[i] <== a[i] + b[i];
    }
}
```

ArrayAdd(3) witness:
a = {1, 2, 3}
b = {6, 5, 4}
out = {7, 7, 7}

ArrayAdd(3) witness:
a = {1, 2, 3}
b = {6, 5, 4}
out = {7, 7, 0}

Unconstrained Output

```
template ArrayAdd(n) {
    signal input a[n];
    signal input b[n];
    signal output out[n];

    for (var i = 0; i < n - 1; i++) {
        out[i] <== a[i] + b[i];
    }
}
```

ArrayAdd(3) witness:
a = {1, 2, 3}
b = {6, 5, 4}
out = {7, 7, 7}

ArrayAdd(3) witness:
a = {1, 2, 3}
b = {6, 5, 4}
out = {7, 7, 0}

Attacker may arbitrarily manipulate the last value of output array!

Veridise. | <-- Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    in*out === 0;
}
```

Veridise. | <-- Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    in*out === 0;
}
```

isZero() witness:
in = 0
out = 1

Veridise. | <-- Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    in*out === 0;
}
```

isZero() witness:
in = 0
out = 1

isZero() witness:
in = 0
out = 0

Veridise. | <-- Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    in*out === 0;
}
```

isZero() witness:
in = 0
out = 1

isZero() witness:
in = 0
out = 0

Must be careful to ensure constraint matches!

Veridise. | Finite Field Size

```
template toBase256(l) {
    signal input in;
    signal output out[l];

    var value = in;
    for (var i = 0; i < l; i++) {
        out[i] <-- value & 255;
        value = value \ 256;
    }

    component bitchecks[l];
    for (var i = 0; i < l; i++) {
        bitchecks[i] = Num2Bits(8); // out[i] is in [0, 256)
        bitchecks[i].in <== out[i];
    }

    signal acc[l];
    for (var i = l-1; i >= 0; i--) {
        if (i == l-1) {
            acc[i] <== out[i];
        } else {
            acc[i] <== 256 * acc[i+1] + out[i];
        }
    }
    acc[0] === in; // acc[0] = out[0] + 256*out[1] + .. + 256^{l-1}*out[l-1]
}
```

Finite Field Size

```
template toBase256(l) {
    signal input in;
    signal output out[l];

    var value = in;
    for (var i = 0; i < l; i++) {
        out[i] <-- value & 255;
        value = value \ 256;
    }

    component bitchecks[l];
    for (var i = 0; i < l; i++) {
        bitchecks[i] = Num2Bits(8); // out[i] is in [0, 256)
        bitchecks[i].in <== out[i];
    }

    signal acc[l];
    for (var i = l-1; i >= 0; i--) {
        if (i == l-1) {
            acc[i] <== out[i];
        } else {
            acc[i] <== 256 * acc[i+1] + out[i];
        }
    }
    acc[0] === in; // acc[0] = out[0] + 256*out[1] + .. + 256^{l-1}*out[l-1]
}
```

toBase256() witness:

in = 0

out = [0, …, 0]

acc[0] = 0

Finite Field Size

```
template toBase256(l) {
    signal input in;
    signal output out[l];

    var value = in;
    for (var i = 0; i < l; i++) {
        out[i] <-- value & 255;
        value = value \ 256;
    }

    component bitchecks[l];
    for (var i = 0; i < l; i++) {
        bitchecks[i] = Num2Bits(8); // out[i] is in [0, 256)
        bitchecks[i].in <== out[i];
    }

    signal acc[l];
    for (var i = l-1; i >= 0; i--) {
        if (i == l-1) {
            acc[i] <== out[i];
        } else {
            acc[i] <== 256 * acc[i+1] + out[i];
        }
    }
    acc[0] === in; // acc[0] = out[0] + 256*out[1] + .. + 256^{l-1}*out[l-1]
}
```

toBase256() witness:

in = 0

out = [0, …, 0]

acc[0] = 0

toBase256() witness:

in = 0

out = encode(p)

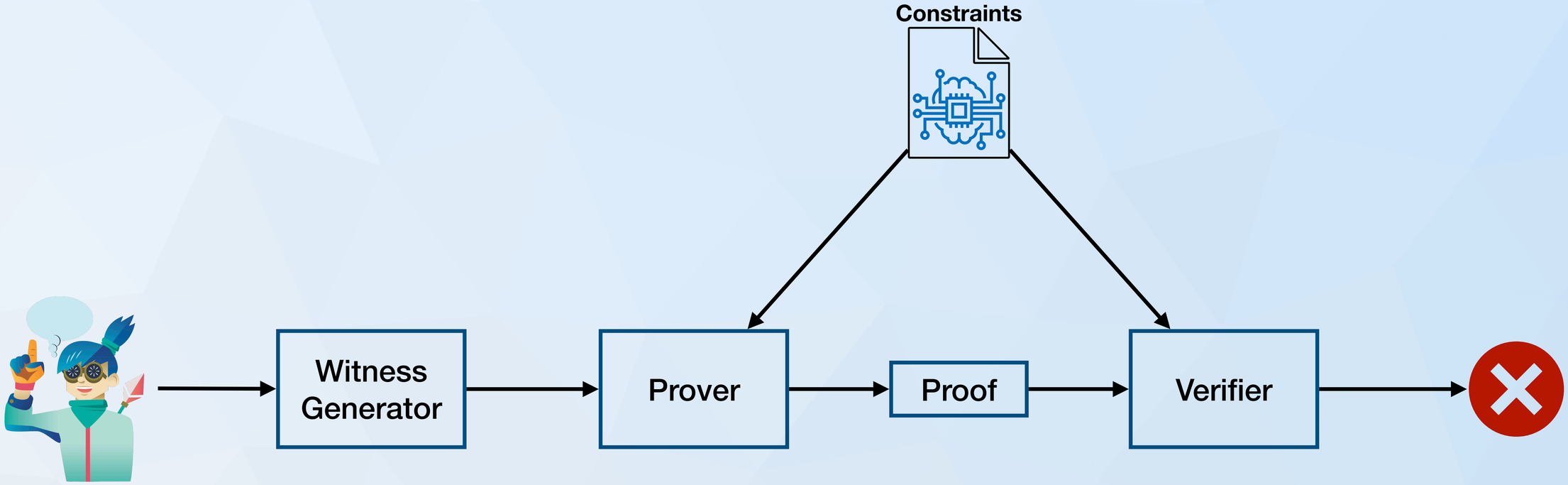
acc[0] = 0

Finite Field Size

```
template toBase256(l) {
    signal input in;
    signal output out[l];

    var value = in;
    for (var i = 0; i < l; i++) {
        out[i] <-- value & 255;
        value = value \ 256;
    }

    component bitchecks[l];
    for (var i = 0; i < l; i++) {
        bitchecks[i] = Num2Bits(8); // out[i] is in [0, 256)
        bitchecks[i].in <== out[i];
    }

    signal acc[l];
    for (var i = l-1; i >= 0; i--) {
        if (i == l-1) {
            acc[i] <== out[i];
        } else {
            acc[i] <== 256 * acc[i+1] + out[i];
        }
    }
    acc[0] === in; // acc[0] = out[0] + 256*out[1] + .. + 256^{l-1}*out[l-1]
}
```

Overflows can allow the circuit to accept n and n + p!

Veridise. | Overconstrained Circuits



Veridise. | === Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    out === 0;
}
```

Veridise. | === Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    out === 0;
}
```

***out* will always be 0!**

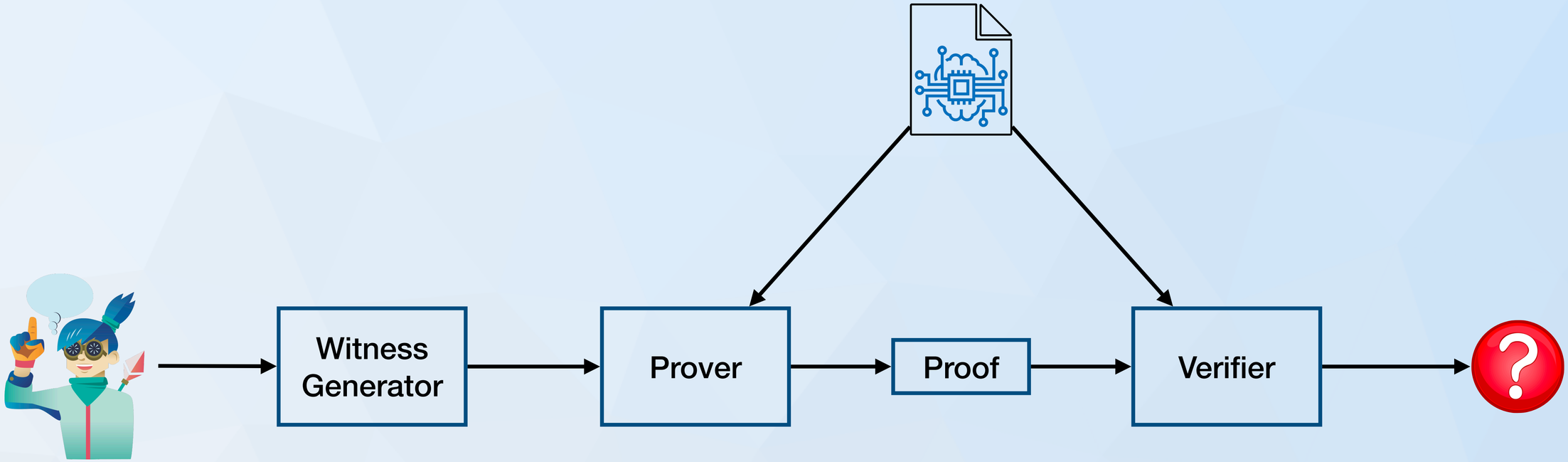
Veridise. | === Operator

```
template isZero() {
    signal input in;
    signal output out;

    out <-- in == 0 ? 1 : 0;
    out === 0;
}
```

***out* will always be 0!**

Must be careful to ensure constraint matches!

Other Bugs



Veridise. | Variable Shadowing

```
function numberOfBits(a) {
      var n = 1;
      var r = 0;
      while (n - 1 < a) {
          var r = r + 1;
          n *= 2;
      }
      return r;
}
```

Variable Shadowing

```
function numberOfBits(a) {
      var n = 1;
      var r = 0;
      while (n - 1 < a) {
          var r = r + 1;
          n *= 2;
      }
      return r;
}
```

Will Always return 0!

Variable Shadowing

```
function numberOfBits(a) {
      var n = 1;
      var r = 0;
      while (n - 1 < a) {
          var r = r + 1;
          n *= 2;
      }
      return r;
}
```

Will Always return 0!

Variable shadowing can cause programming errors!

Veridise. | Logical Errors

```
template withdraw(n) {
    assert(n <= 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2Bits(n); // assert (bal < 2^n)
    n2b1.in <== bal;
    component n2b2 = Num2Bits(n); // assert (amt < 2^n)
    n2b2.in <== amt;
    component lt = LessThan(n); // check amt < bal
    lt.in[0] <== bal;
    lt.in[1] <== amt;

    out <== bal - amt;
}
```

Logical Errors

```
template withdraw(n) {
    assert(n <= 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2Bits(n); // assert (bal < 2^n)
    n2b1.in <== bal;
    component n2b2 = Num2Bits(n); // assert (amt < 2^n)
    n2b2.in <== amt;
    component lt = LessThan(n); // check amt < bal
    lt.in[0] <== bal;
    lt.in[1] <== amt;

    out <== bal - amt;
}
```

Withdraw(252) witness:

amt = 4

bal = 5

out = 1

Logical Errors

```
template withdraw(n) {
    assert(n <= 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2
    n2b1.in <== bal;
    component n2b2 = Num
    n2b2.in <== amt;
    component lt = LessThan(n); // check amt < bal
    lt.in[0] <== bal;
    lt.in[1] <== amt;

    out <== bal - amt;
}
```

```
sert (bal < 2^n)

sert (amt < 2^n)
```

Missing constraint
lt.out === 0

Withdraw(252) witness:
amt = 4
bal = 5
out = 1

Withdraw(252) witness:
amt = 6
bal = 5
out = -1

Logical Errors

```
template withdraw(n) {
    assert(n <= 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2
    n2b1.in <== bal;
    component n2b2 = Num
    n2b2.in <== amt;
    component lt = LessThan(n); // check amt < bal
    lt.in[0] <== bal;
    lt.in[1] <== amt;

    out <== bal - amt;
}
```

```
sert (bal < 2^n)
sert (amt < 2^n)
```

Missing constraint
lt.out === 0

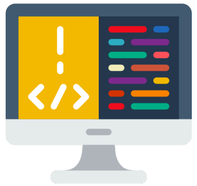
Can be simple programming mistakes or behavioral mistakes

Veridise. | Avoiding Bugs

Testing

✓ No Specialty Knowledge
✓ Quick Feedback
X No Guarantees

Tools



✓ Some Guarantees
✓ Little Specialty Knowledge
X Possibly Long Runtimes
X Weak Guarantees

Auditing

✓ No Specialty Knowledge (For Developer)
✓ Expert Reviewer
✓ Capable of Complex Logical Reasoning
X Significant Effort + Wait
X Varying Quality
X No Guarantees

Verification

✓ Strong Guarantees
X Significant Specialty Knowledge
X Significant Effort + Wait

Veridise. | Circomspect

Helps find common programming errors!

Issues Found

Detects:

- Variable Shadowing
- Dead Variable
- Use of <-- over <== for quadratic expressions
- Dead Branch
- Code Complexity
- Unconstrained use of Num2Bits and Bits2Num
- Informational hints for manual inspection

Veridise. | Circomspect

Pros	Cons
✓ Lightweight	X Shallow
✓ Requires No User Input	X No Custom Properties
✓ No Specialty Knowledge	X False Positives

Veridise. | Picus

1. Static Analysis of Constraints (ECNE)

Apply predefined rules to quickly detect if circuit is properly constrained

$$\begin{cases} \text{input } x \\ \text{output } y \\ z = 3x + 4 \\ y = z + 2x \end{cases}$$

Since y is linear in x, z we immediately infer it is not under constrained

SMT Solver

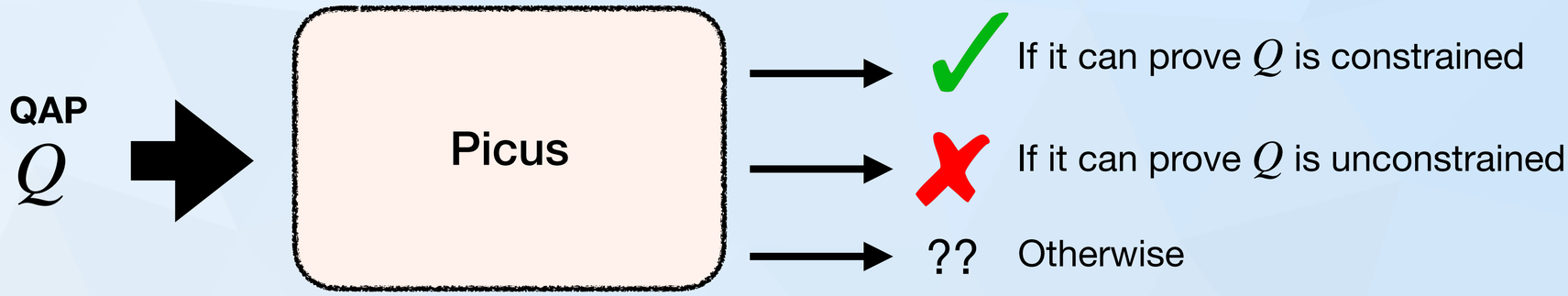
Underconstrained can be expressed as SMT query

$$\exists y_1, y_2 \,.\, Q[y_1/y] \wedge Q[y_2/y] \wedge y_1 \neq y_2$$

SAT means the circuit is underconstrained

Strategy	Pros	Cons
ECNE	**Scaleable**	Many False Positives
SMT	**Precise**	Can’t Scale

Veridise. | Picus



Combine the strengths of Static Analysis and SMT!

Static Analysis and SMT phases interact in a loop

Veridise. | Picus

Pros	Cons
✓ Requires No User Input	X No Custom Properties
✓ No Specialty Knowledge	X Potentially Long Runtimes

Veridise. | Questions?

Follow us on Twitter!

@VeridiseInc