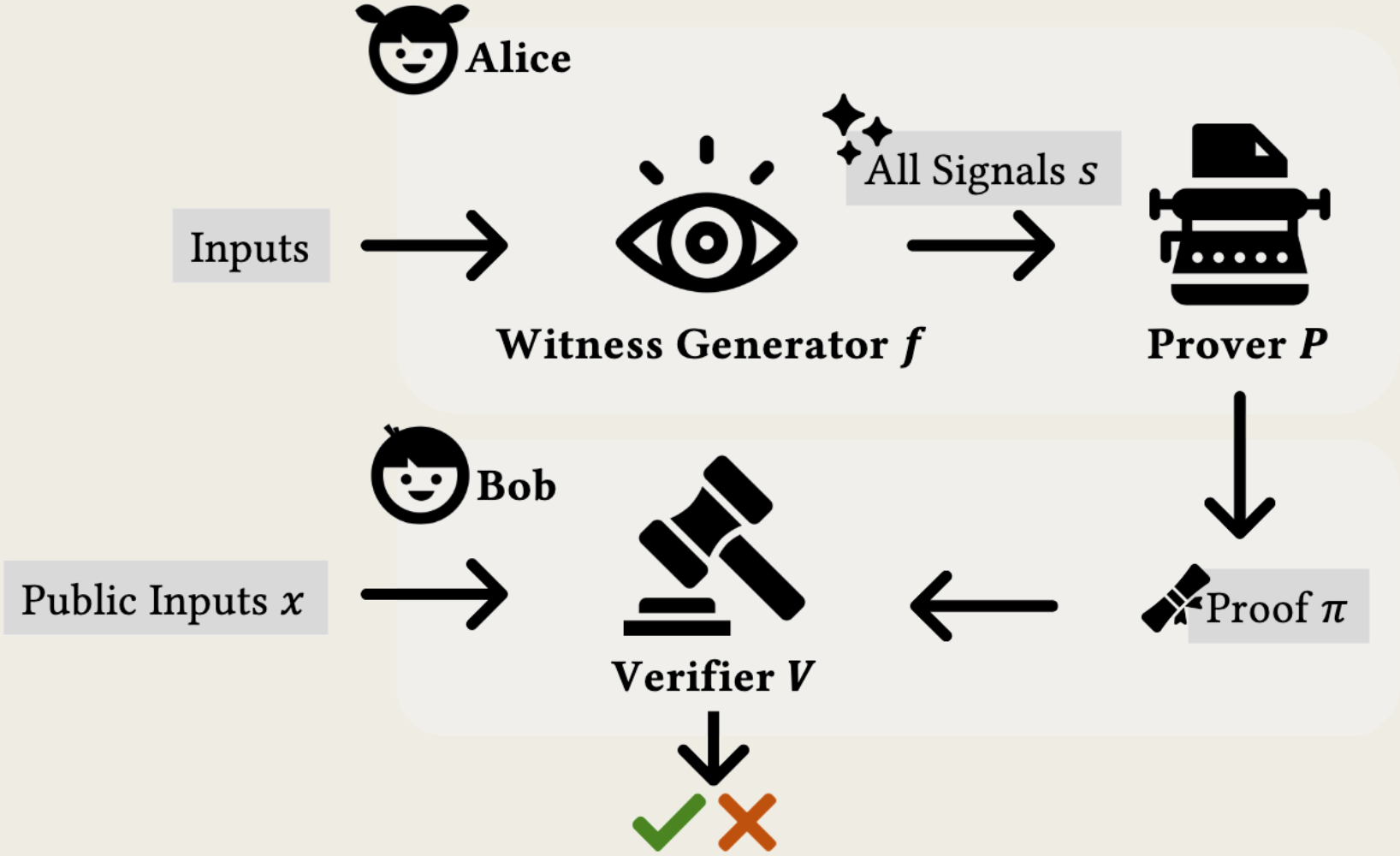
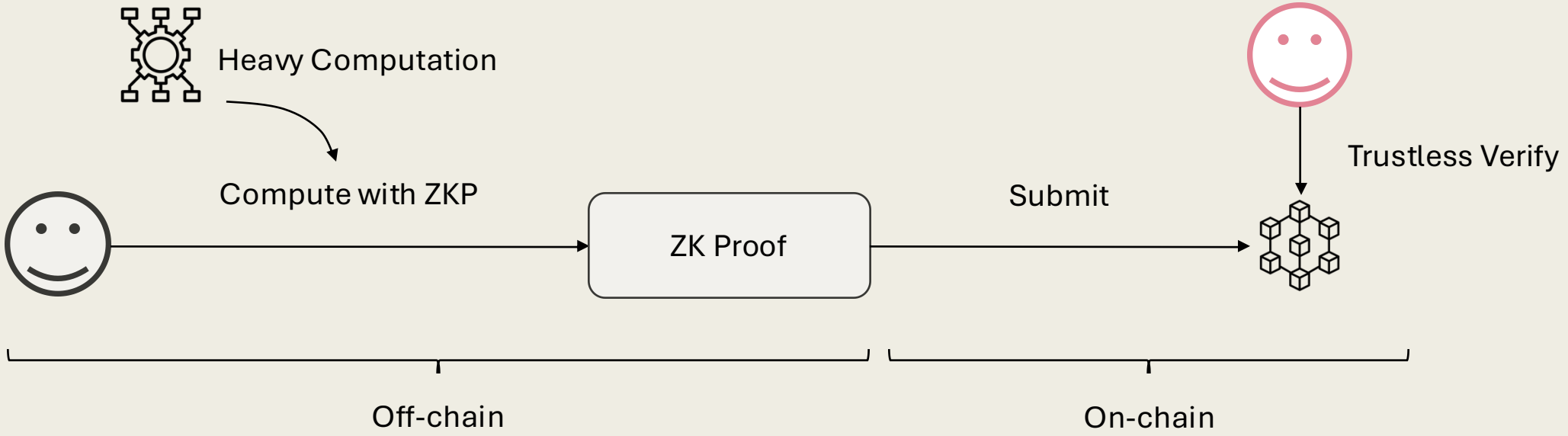
PRACTICAL SECURITY ANALYSIS OF ZERO-KNOWLEDGE PROOF CIRCUITS

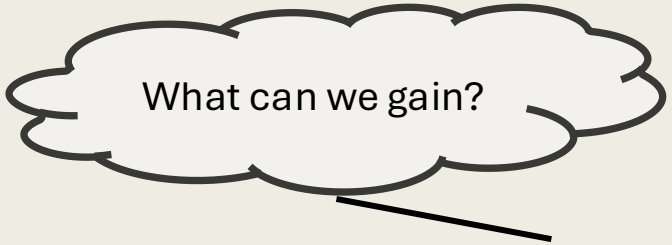
Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles,
Shankara Pailoor, Kyle Charbonnet, Isil Dillig, Yu Feng

Zero-knowledge Proofs (ZKPs)



ZKP Technologies & Blockchain

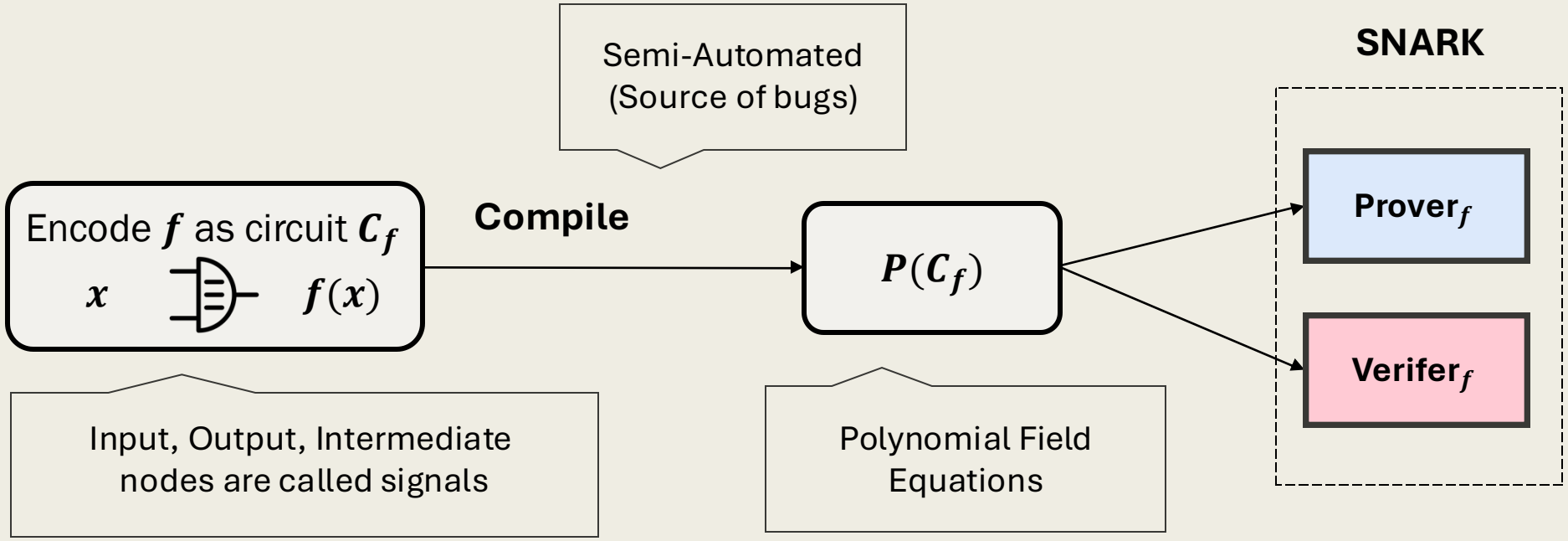




Scalability

SNARK Workflow

Suppose we want to construct a SNARK for $f(x)$



Semi-Automated Compilation

1. Some computation not expressible as field equations

pointbits.circom

```
template Bits2Point_String() {
signal input in[256];
signal output out[2];

// . . .
var tmp = (1 - y2)/(a - 2*y2);
var x = sqrt(tmp)
}
```

Can't be expressed as field equations

Required Manual Constraint

$$x^2 = tmp$$

2. Highly non-trivial to automatically infer constraints

Significance

Ideally Field Equations and Circuit should be equivalent:

For every $x, z. C_f(x) = z$ if and only if x, z satisfy $P(C_f)$.

Circuits and equations are often not equivalent!

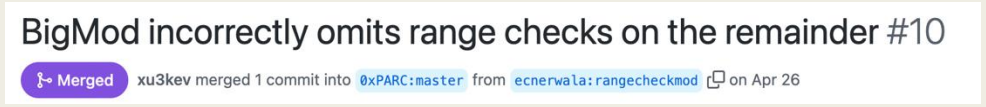
Manually adding constraints is
time consuming and error prone

Under-constrained Circuits

Circuit is under-constrained if the output signals are not uniquely determined by the input signals in polynomial form.

Allows attackers to get bogus proofs verified.

Be used to drain all tokens



Be used to double-spend

Under-constrained Circuits

(Constraint-Computation Discrepancies)

```
template Edwards2Montgomery() {
    signal input in[2];
    signal output out[2];
    out[0] <-- (1 + in[1]) / (1 - in[1]);
    out[1] <-- out[0] / in[0];
    out[0] * (1-in[1]) === (1 + in[1]);
    out[1] * in[0] === out[0];
}
component main = Edwards2Montgomery();
```

Computation (Prover)

```
out[0] <-- (1 + in[1]) / (1 - in[1]);
out[1] <-- out[0] / in[0];
```

Constraints (Verifier)

```
out[0] * (1-in[1]) === (1 + in[1]);
out[1] * in[0] === out[0];
```

```
out[0] === 0;
1 + in[1] === 0;
in[0] === 0;
```

Any `out[1]` could bypass the verification!

Computation

```
out[0] <-- 0 / 2;
out[1] <-- 0 / 0; (Undefined)
```

Constraints

```
0 * 2 === 0;
out[1] * 0 === 0;
```

Circuit Dependence Graph (CDG)

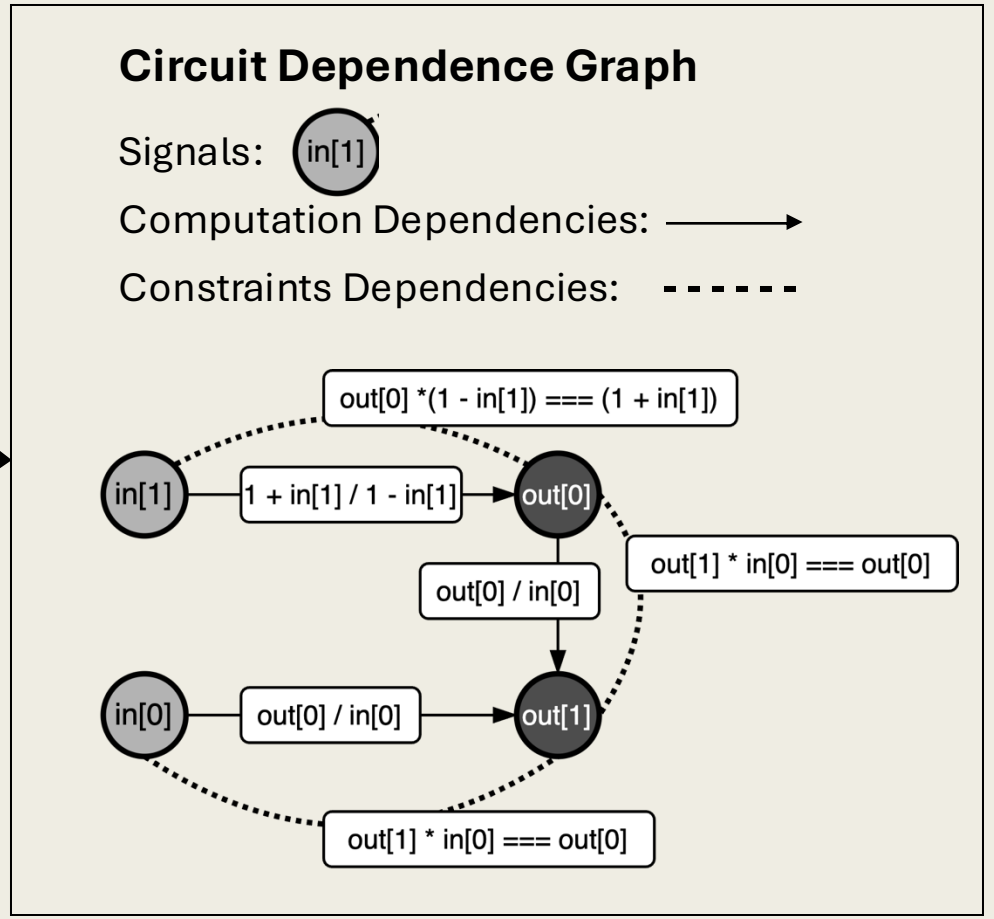
Signals and computation and constraints dependencies between them compound a circuit:

Computation

```
out[0] <-- (1 + in[1]) / (1 - in[1]);
out[1] <-- out[0] / in[0];
```

Constraints

```
out[0] * (1-in[1]) === (1 + in[1]);
out[1] * in[0] === out[0];
```

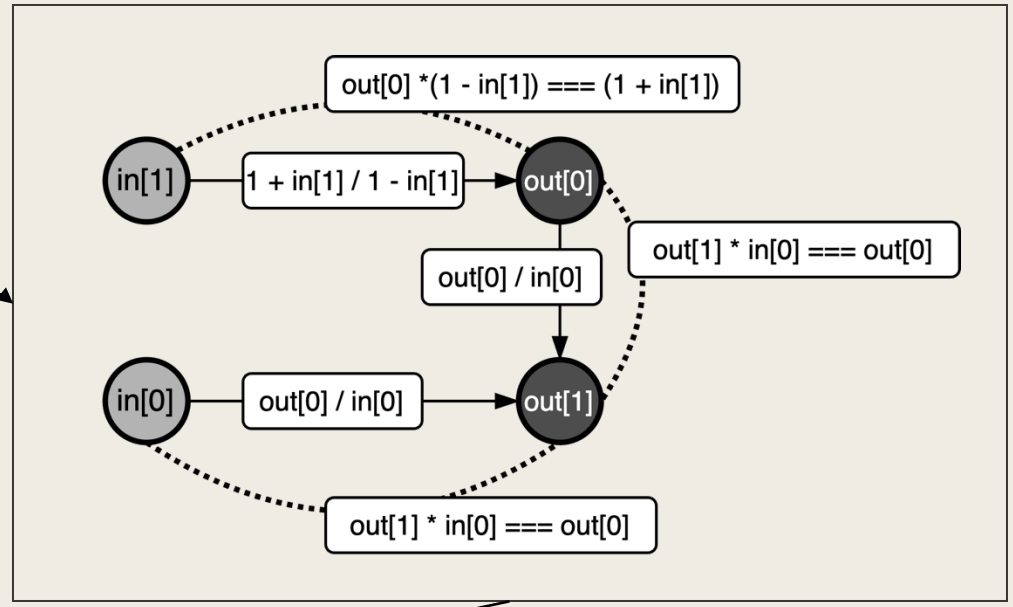


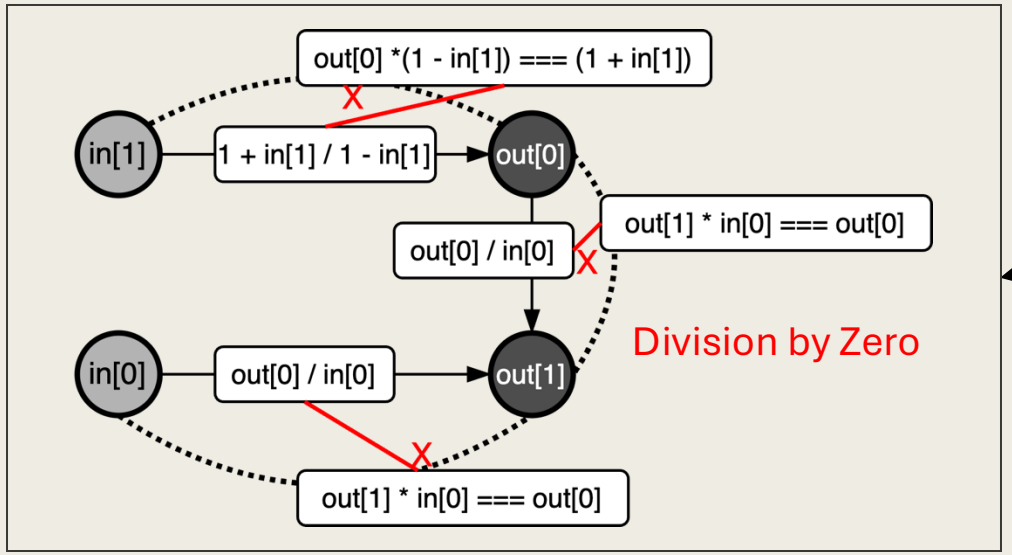
Bug Detection via CDG

Circom code

```
template Edwards2Montgomery() {
    signal input in[2];
    signal output out[2];
    out[0] <-- (1 + in[1]) / (1 - in[1]);
    out[1] <-- out[0] / in[0];
    out[0] * (1-in[1]) === (1 + in[1]);
    out[1] * in[0] === out[0];
}
component main = Edwards2Montgomery();
```

CDG





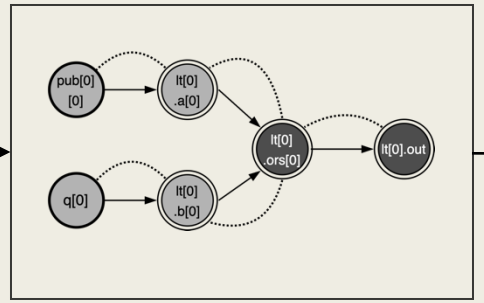
Bug Detection via CDG (Automatic)

Circom code

```
template CoreVerifyPubkeyG1(n, k){ // ...
    component lt[10];
    for(var i=0; i<10; i++){
        lt[i] = BigLessThan(n, k);
        for(var idx=0; idx<k; idx++) {
lt[i].b[idx] <== q[idx];}
    }
    for(var idx=0; idx<k; idx++){
lt[0].a[idx] <== pub[0][idx];}
         // lt.out is not used
}
component main = CoreVerifyPubkeyG1(55,7);
```

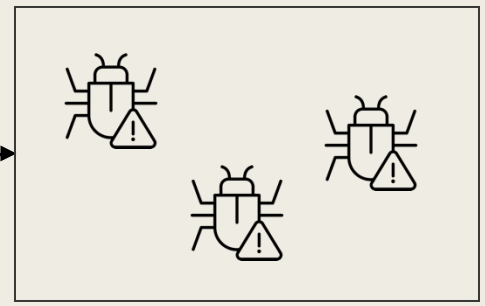
CDG Construction

CDG



VDL Query

Reported Bugs



Major sources of bugs

1. Non-deterministic Signals
2. Unsafe Component Usage
3. Constraint-Computation Discrepancies

Detectors written in VDL

Tips: VDL is ` Vulnerability Description Language`.

Evaluation Results

- We implemented 9 detectors for the bug categories we mentioned.
- We collected 258 Circom circuits from 17 popular open-source projects.
- We inspected results manually to distinguish actual vulnerabilities and false alarms.
- ZKAP had a lower FP rate.
- ZKAP found 81 vulnerabilities across all bug categories automatically.
- ZKAP identified previously unknown bugs, which were confirmed and fixed by developers.

Tips: ZKAP is the name of our tool: GitHub Codebase

Thanks and Q&A