CirC

Compiler Infrastructure for Cryptosystems and Verification

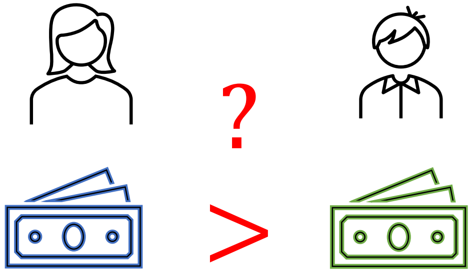
Alex Ozdemir, Fraser Brown, Riad S. Wahby

Pronunciation: SIR-see

Cryptosystems are computers

53+/103 CRYPTO'21 papers!

Multi-Party Computation



$\boldsymbol{f}(x, y)$

Fully Homomorphic Encryption

$\mathrm{Enc}(x) \rightarrow \mathrm{Enc}(\boldsymbol{f}(x))$

Zero-Knowledge

prove that

$\boldsymbol{f}(x) = 1$

functionality and *performance* depend on $\boldsymbol{f}$'s representation

Cryptosystems are low-level

Problems:

1. How do we *express* f ?

2. How do we *optimize* f ?

4 bit comparison

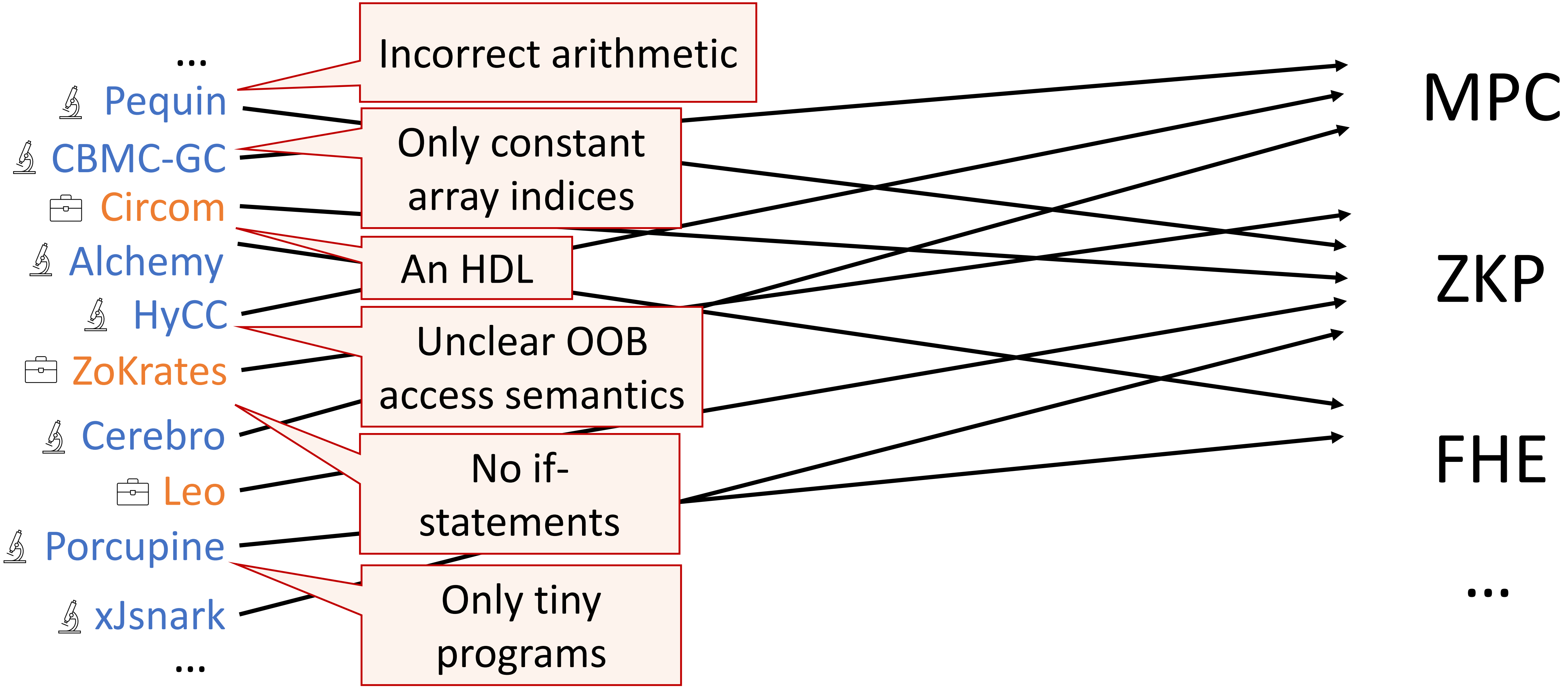
f:

```
a0(a0 - 1) = 0 ; a1(a1 - 1) = 0 ;
a2(a2 - 1) = 0 ; a3(a3 - 1) = 0 ;
b0(b0 - 1) = 0 ; b1(b1 - 1) = 0 ;
b2(b2 - 1) = 0 ; b3(b3 - 1) = 0 ;
a0(b0 - 1) = w0 ; 2a1(b1) = e1 -
a1 - b1; a0(b0 - 1) = w0 ; ....
```

f:

```
a0(a0 - 1) = 0
; a1(a1 - 1) =
0 ; a2(a2 - 1)
= 0 ; a3(a3 -
1) = 0 ; ....
```

Cryptosystems need compilers



[SBVBPW’13][BFRSBW’13][FHKSV’14][WSRBW’15][CroPeiSha’18][BDKKS’18][EbeTai+Sch’19][ZDCP’21] …

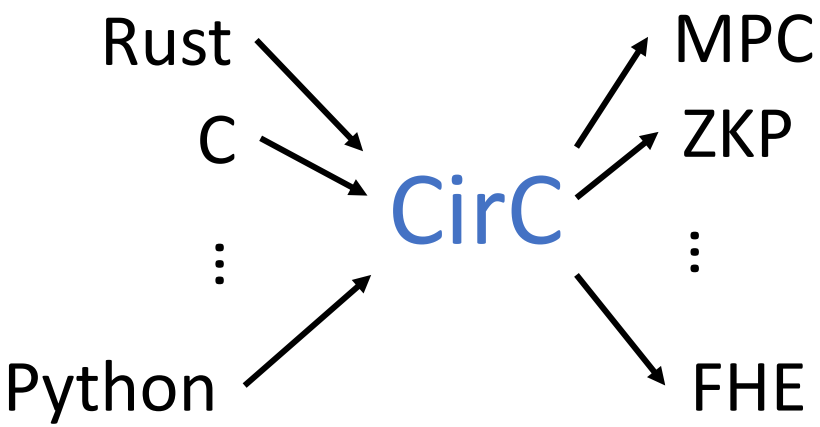
Can we share infrastructure?

Benefits:

1. Save time
   → more research
   → better product

2. Produce better compilers

This Talk: CirC



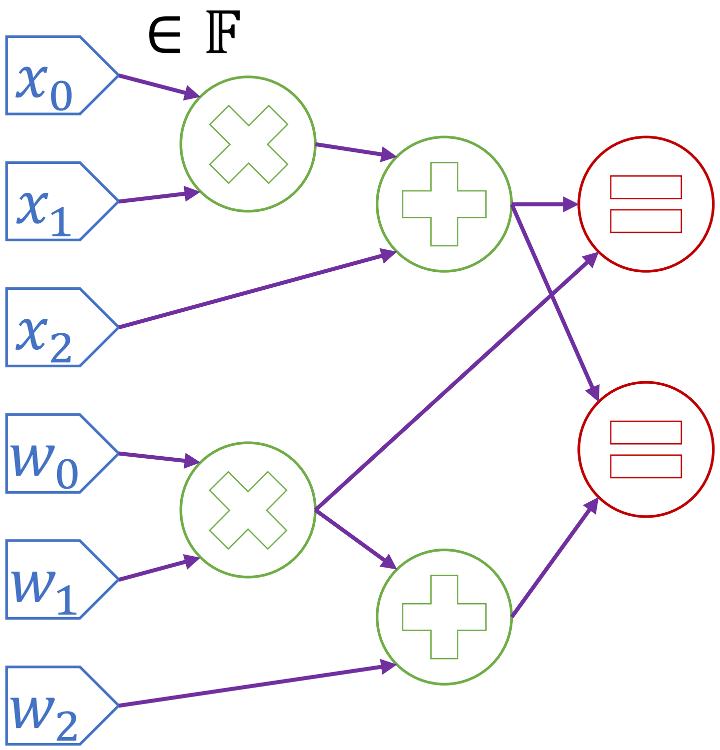
1. Design & computational model
2. Output performance
3. Extensibility
4. Cross-over opportunities

CirC: Compiler Infrastructure for Cryptosystems and Verification

Cryptosystems process arithmetic circuits

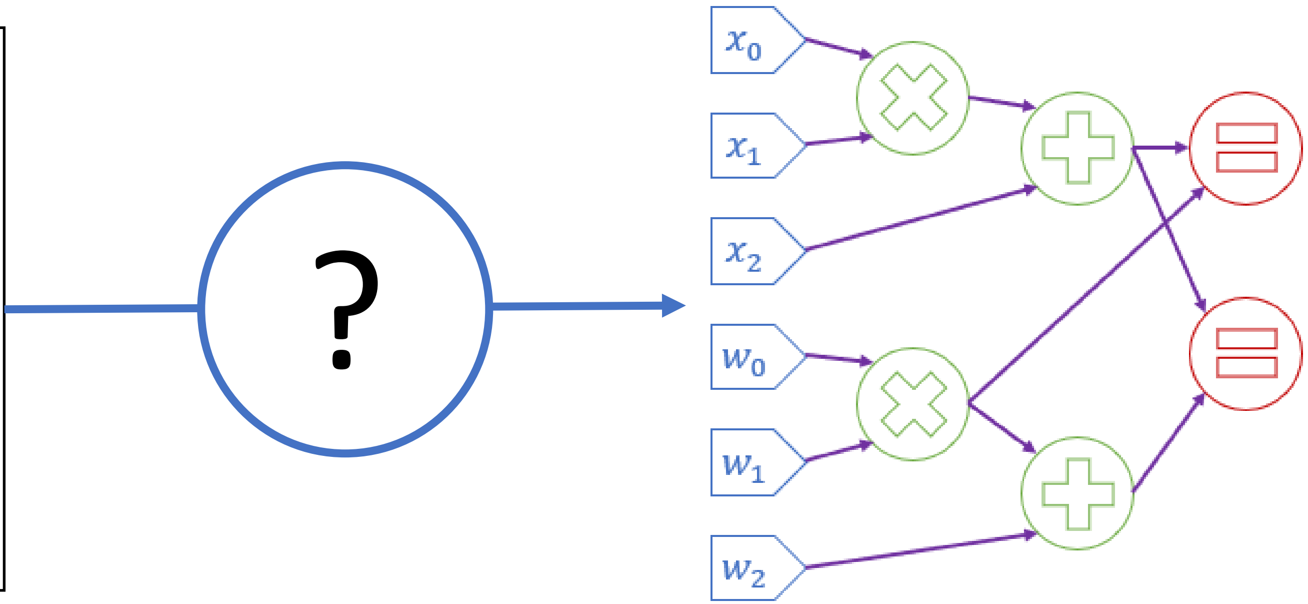
- Directed acyclic graph
  - inputs
  - wires
  - gates
- Domain: finite field
  - wires → field values
  - gates → field operations
- Optional equality assertions
- Fewer gates → better perf

$\boldsymbol{f}(\vec{x}, \vec{w}) \approx$



How do we compile programs to circuits?

```
x = y
x = x * y
if y > 0:
  x = x * x

  for (z=1;z<3;z++):
    y = y + z
```



1. Eliminate mutation

```
x = y
x = x * y
```

$\rightarrow$

$$x_1 = y_1$$
$$x_2 = x_1 * y_1$$

2. Eliminate branches

```
x = y
x = x * y
if y > 0:
  x = x * x
```

$x_1 = y_1$
$x_2 = x_1 * y_1$
$x_3 = y_1 > 0$
$\quad ?\ x_2 * x_2$
$\quad :\ x_2$

C “ternary” operator: a MUX.

3. Unroll Loops

```
x = y
x = x * y
if y > 0:
  x = x * x

  for (z=1;z<3;z++):
    y = y + z
```

$x_1 = y_1$
$x_2 = x_1 * y_1$
$x_3 = y_1 > 0$
$\quad ?\ x_2 * x_2$
$\quad :\ x_2$
$z_1 = 1$
$y_2 = y_1 > 0\ \&\&\ z_1 < 3$
$\quad ?\ y_1 + z_1$
$\quad :\ y_1$
$z_2 = z_1 + 1$
$y_3 = y_1 > 0\ \&\&\ z_1 < 3\ \&\&\ z_2 <$
$\quad ?\ y_2 + z_2$

How do we compile programs to circuits?

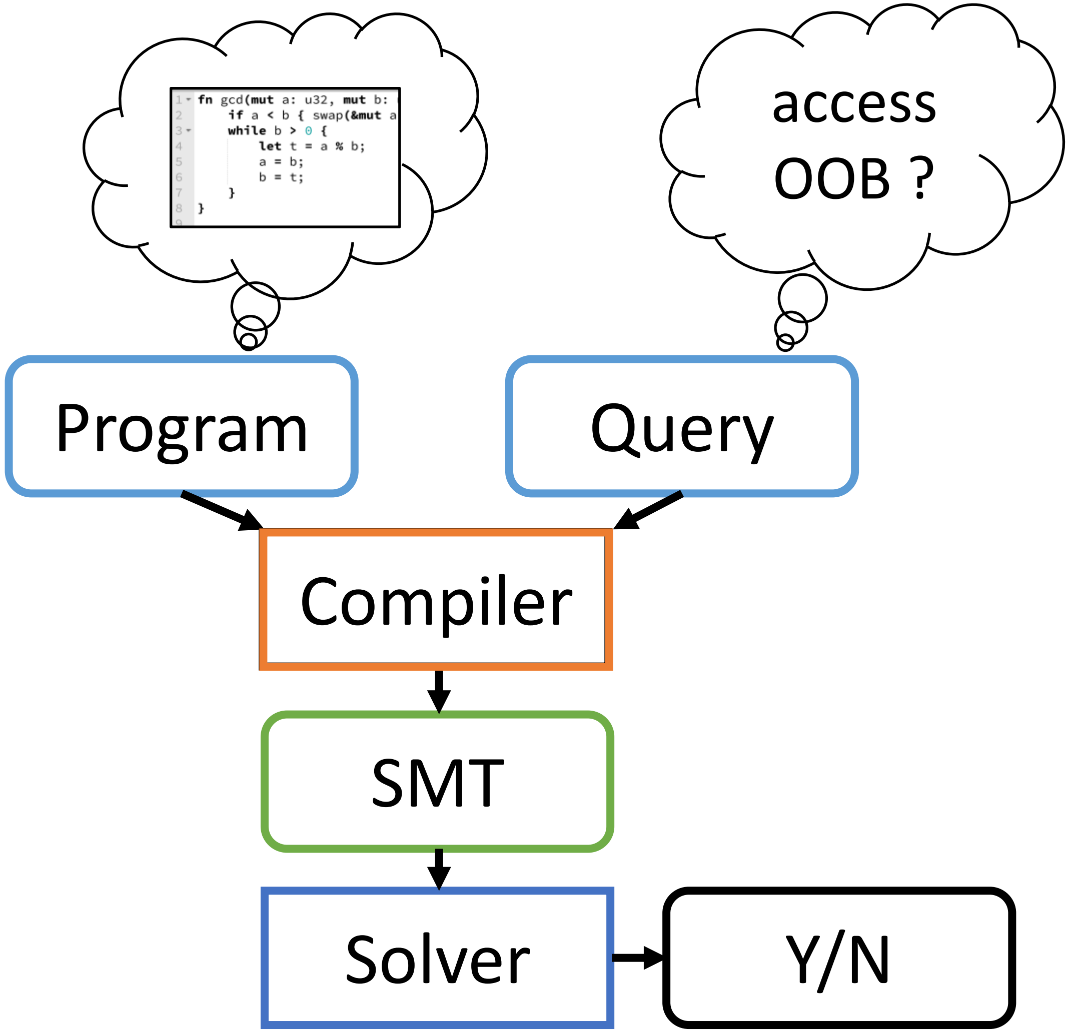
1. Eliminate mutation
2. Eliminate branches
3. Unroll loops
4. Inline function calls
5. Functionalize arrays
6. …

Similar to
Bounded Model
Checking!

Bounded Model Checking is compilation

1. Model the program* and query as a logical formula

2. Check the model with a solver



**the BMC connection is latent in projects like CMBC-GC

*consider bounded traces

Computational Model:
Existentially Quantified Circuits (EQCs)

SMT	ZKP
FHE	MPC

- Mutation free
- Control-flow free
- Fixed-size (non-uniform)

circuits

- Variables of *restricted knowledge*:
  - ZKP: known to prover
  - FHE: known only to user
  - MPC: distributed knowledge
  - SMT: existentially quantified

existential quantifiers (with metadata)

Connection: QF_* logics in SMT-LIB

Computational Model:
Existentially Quantified Circuits (EQCs)

Problem:
Existentially Quantified Circuits (EQCs)
are not easy to program!

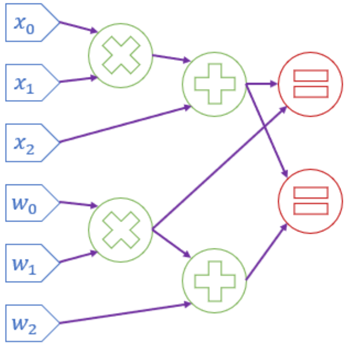
CirC compiles high-level languages to circuits

High-Level Language
- Mutation
- RAM
- Control-Flow
- Uniform

→ **CirC** →

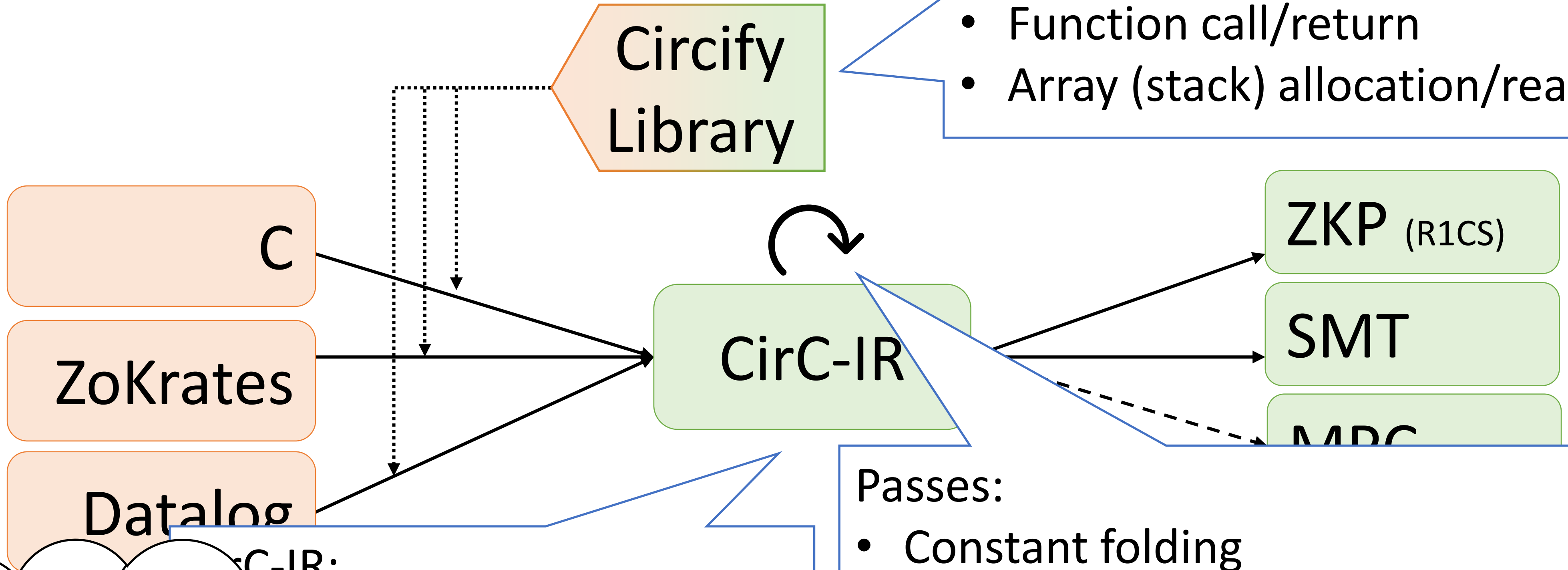
EQC
- No mutation
- No RAM
- No control-flow
- Non-uniform

```
x = y
x = x * y
if y > 0:
  x = x * x

  for (z=1;z<3;z++):
    y = y + z
```



CirC's Architecture

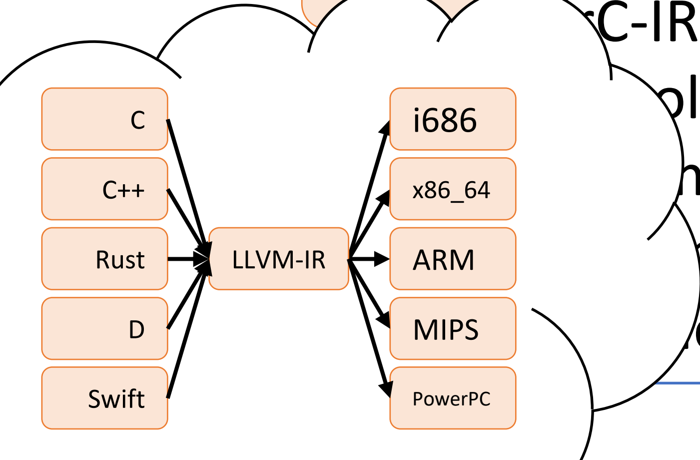
Manages:
- Variable scope/decl./read/write
- Control flow: branch, break,…
- Function call/return
- Array (stack) allocation/read/write



…C-IR:
…oleans, bit-vectors,
…ne fields, floating-point
…nded arrays
…es

harder:
pgm → circ

Passes:
- Constant folding
- Peephole (e.g. bitwise ternary)
- Tuple elimination
- Array elimination (oblivious, linear scan, transcript-checking)

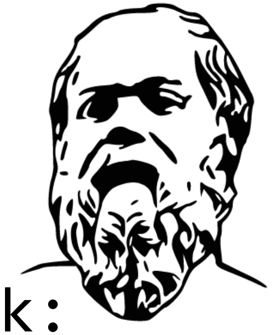


Case Study: ZoKrates to ZKP

With CirC, it’s ***easy*** to build a ***performant*** compiler.

ZoKrates (v0.6.2)

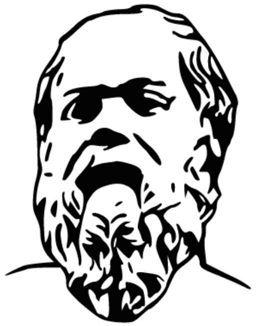
- Targets ZKP
- Types: Booleans, machine-integers, fields, arrays, structures,…
- Good tooling
- Supported by the Ethereum Foundation



Stdlib: ecc/edwardsAdd.zok:

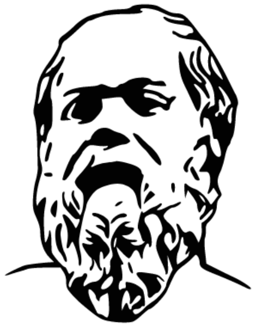
```
from "ecc/babyjubjubParams" import BabyJubJubParams as P

def main(field[2] pt1, field[2] pt2, P pp) -> field[2]:

    field a = pp.JUBJUB_A
    field d = pp.JUBJUB_D

    field u1 = pt1[0]
    field v1 = pt1[1]
    field u2 = pt2[0]
    field v2 = pt2[1]

    field uOut = (u1*v2 + v1*u2) / (1 + d*u1*u2*v1*v2)
    field vOut = (v1*v2 - a*u1*u2) / (1 - d*u1*u2*v1*v2)

    return [uOut, vOut]
```

ZoKrates' reference compiler is big



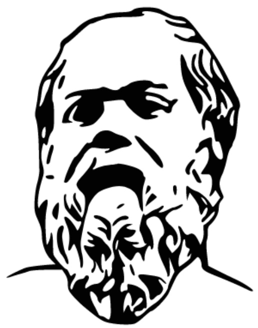
Compiler	Reference	
Lines of Code	~28,000	
Development Time	3 years	
Contributors	36	
Output Size	good	

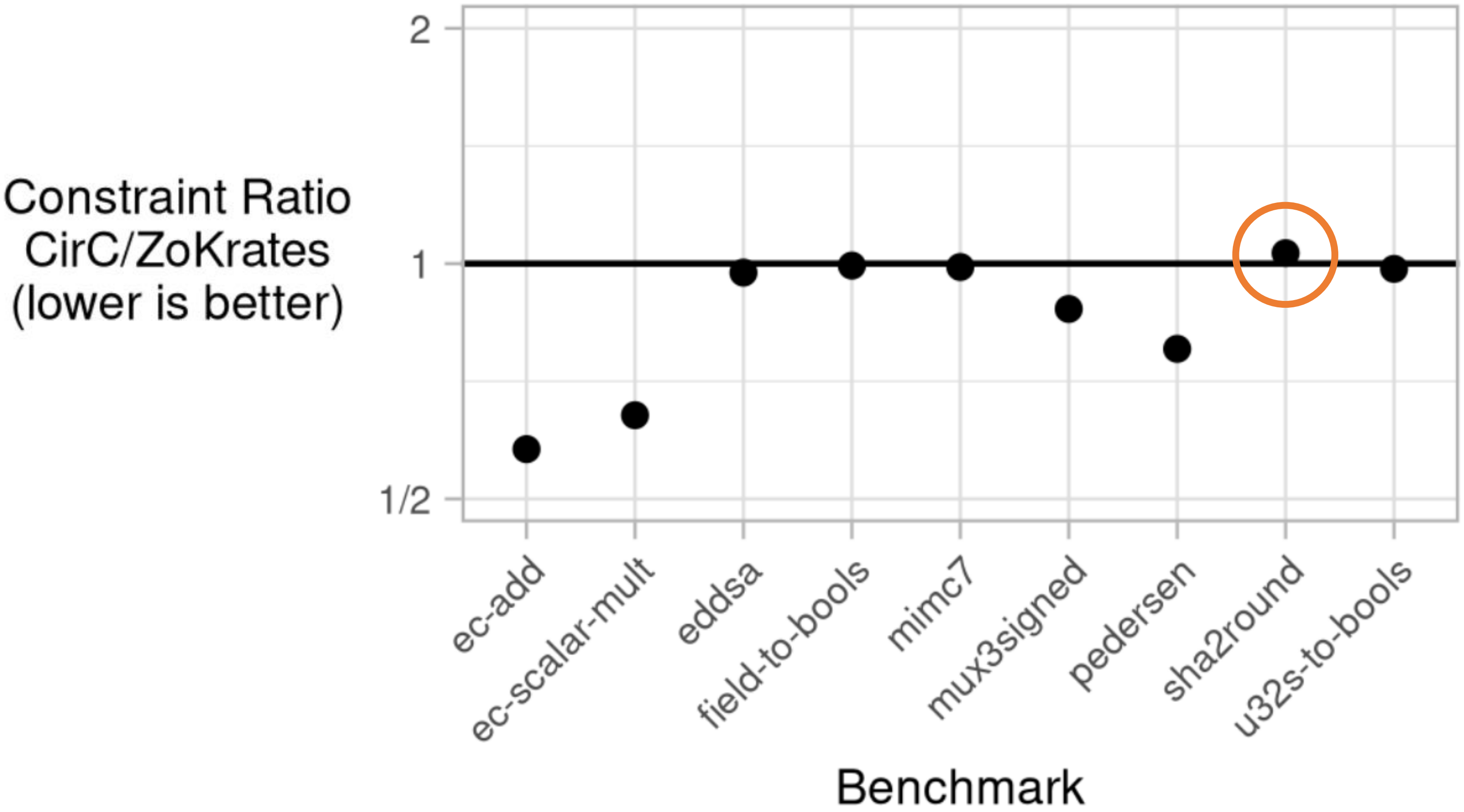
Compiling ZoKrates is easier with CirC

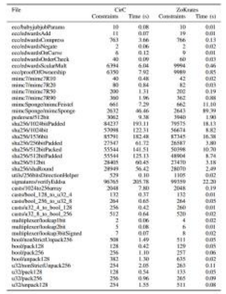


Compiler	Reference	CirC
Lines of Code	~28,000	**~700**
Development Time	3 years	**1 week**
Contributors	36	**1**
Output Size	good	**better**

CirC-ZoKrates outperforms Zokrates







Full comparison in appendix

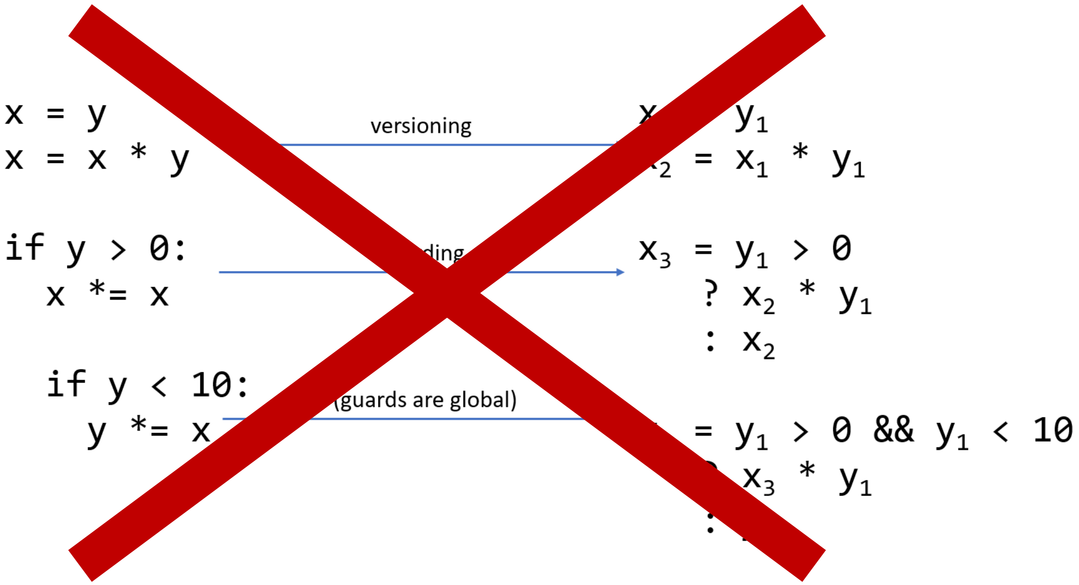
CirC's ZKP compilers perform well

Input Language	Baseline Compiler	CirC Improvement
ZoKrates	ZoKrates	~1x - 1.8x
C	Pequin	~1x - 8x
Circom	Circom	*identical (HDL)*

A Tour of the ZoKrates Front-End

Building a circuit compiler in “three easy pieces”

Writing a New Front-End



Key Steps:

1. Represent language values using CirC-IR
2. Hook into Circify
3. Translate AST (using Circify)

Mapping Language Values to CirC-IR

- All ZoKrates types have CirC-IR analogs:
  - `bool` → booleans
  - `u8`, `u16`, `u32` → bit-vectors
  - `field` → prime fields
  - `[]` → field-indexed arrays
  - structures → tuples

/src/front/zokrates/term.rs

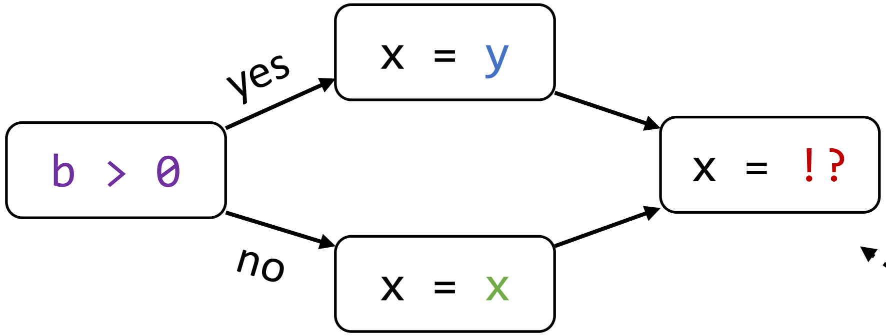
```
pub enum Ty {
    Uint(usize),
    Bool,
    Field,
    Struct(String, FieldList<Ty>),
    Array(usize, Box<Ty>),
}
```

```
pub struct T {
    pub ty: Ty,
    pub term: ir::Term,
}
```

Map values to CirC-IR » Hook into Circify » "Interpret" the AST

Hooking Into Circify

- Circify must *merge states*



/src/front/zokrates/term.rs

```
fn ite(c: Term, a: T, b: T) -> Result<T, String> {
    if &a.ty != &b.ty {
        Err(format!("Cannot perform ITE on {} and {}", a, b))
    } else {
        Ok(T::new(a.ty.clone(), term![Op::Ite; c, a.term, b.term]))
    }
}
```

```
x = Op::Ite(b > 0, y, x)
```

- Circify needs *if-then-else* expressions for language values

“Interpreting” the AST Using Circify

/src/front/zokrates/mod.rs*

```
impl Interp {
    fn stmt(&mut self, s: &ast::Statement<'ast>) {                    // "interpret" a statement
```

*w/o error handling

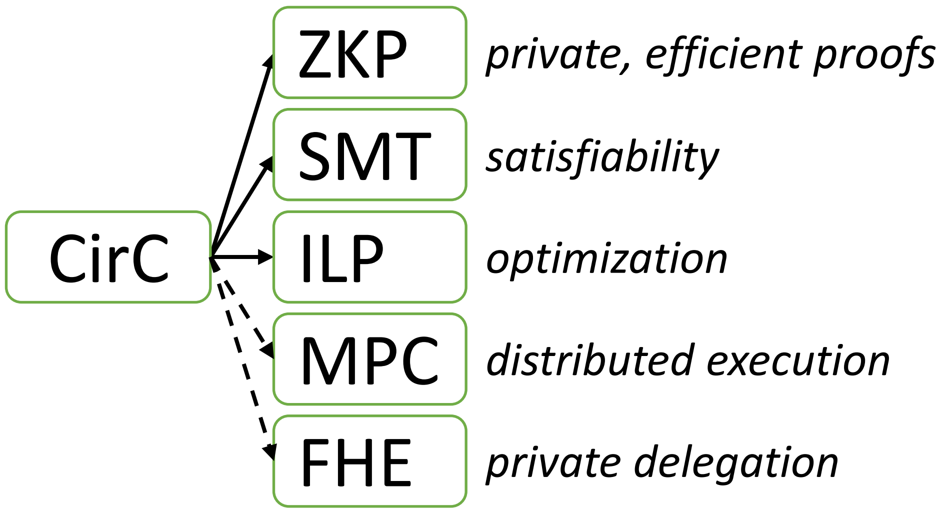
Map values to CirC-IR ⋙ Hook into Circify ⋙ “Interpret” the AST

Stretch break!

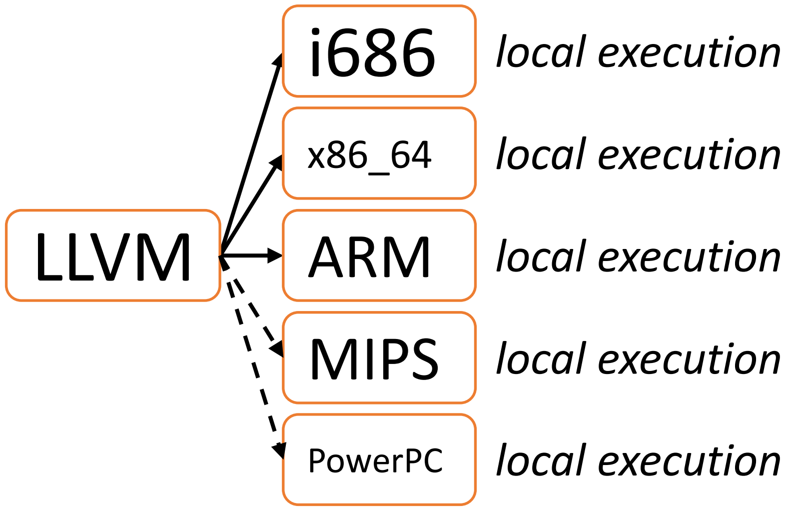
Cross-Over Opportunities

Could applications use *multiple* targets?

CirC: targets have *different* purposes

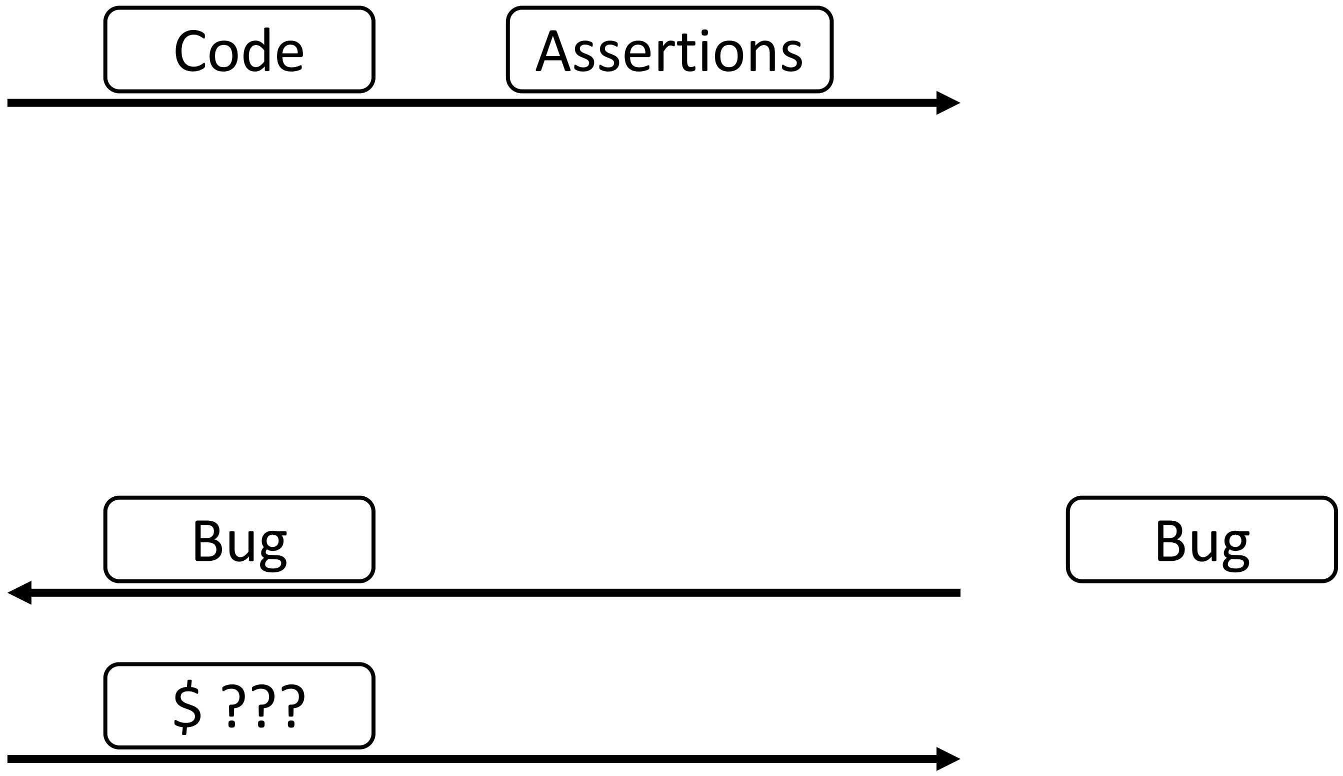


LLVM: targets have the same purpose

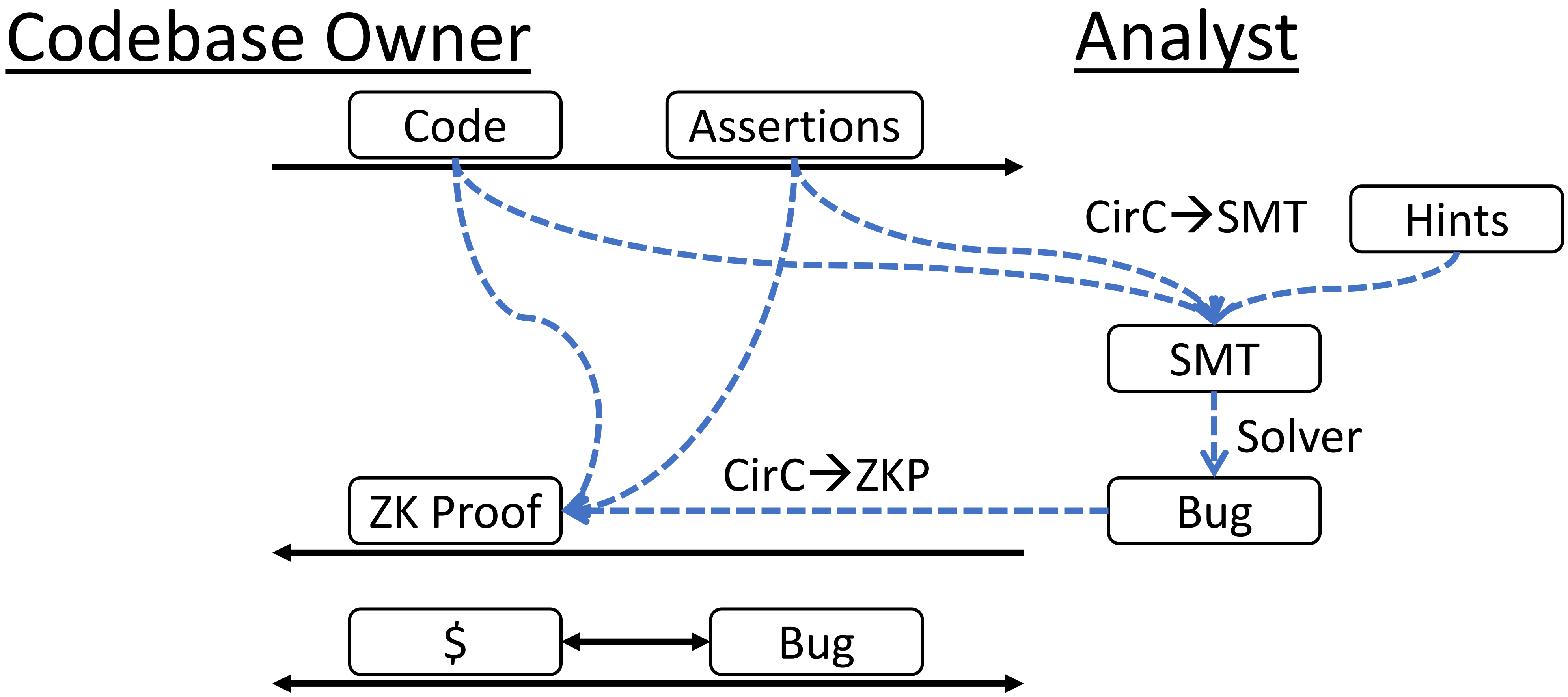


Supporting ZK Bug Bounties

Codebase Owner Analyst



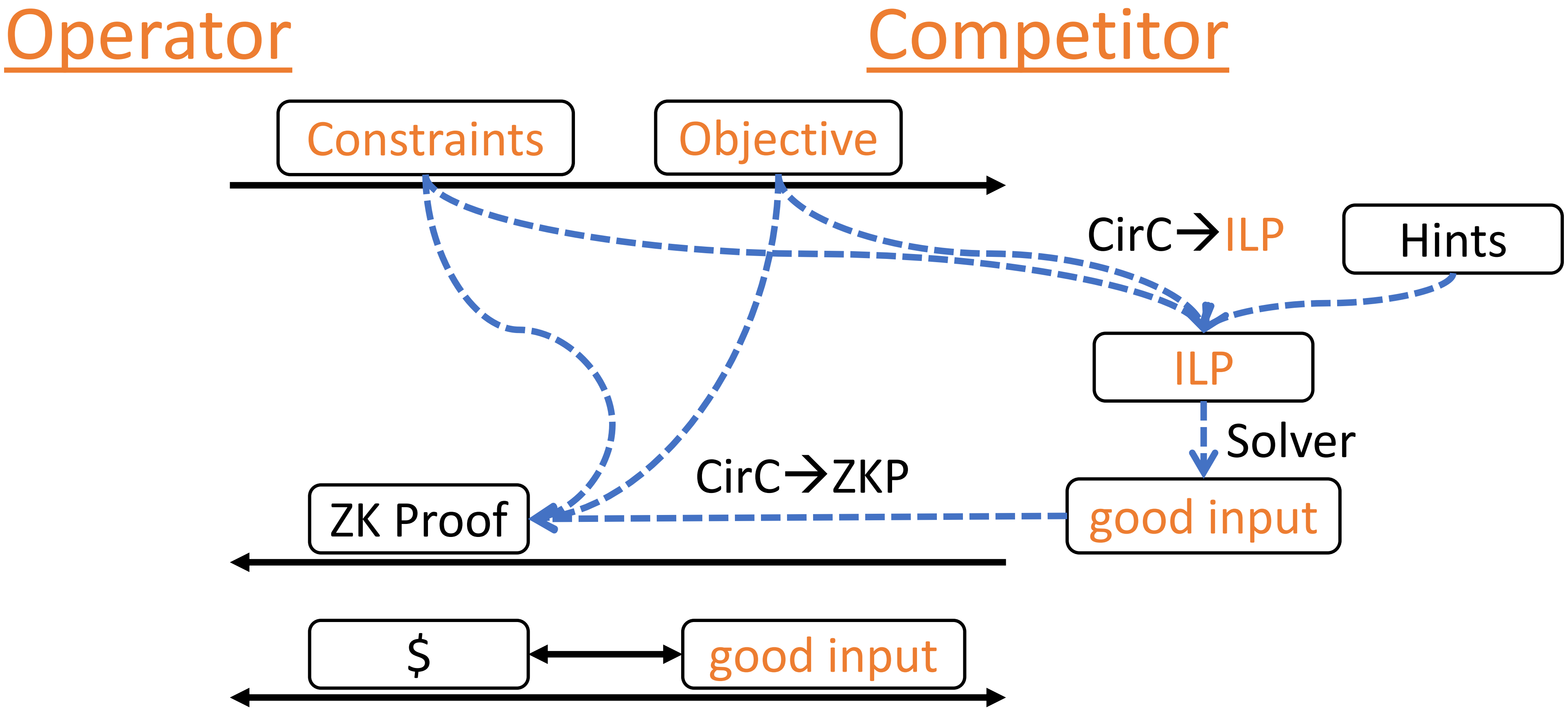
Supporting ZK Bug Bounties



Proof of concept: CVE-2014-3570 (OpenSSL)

≈ 60 LOC

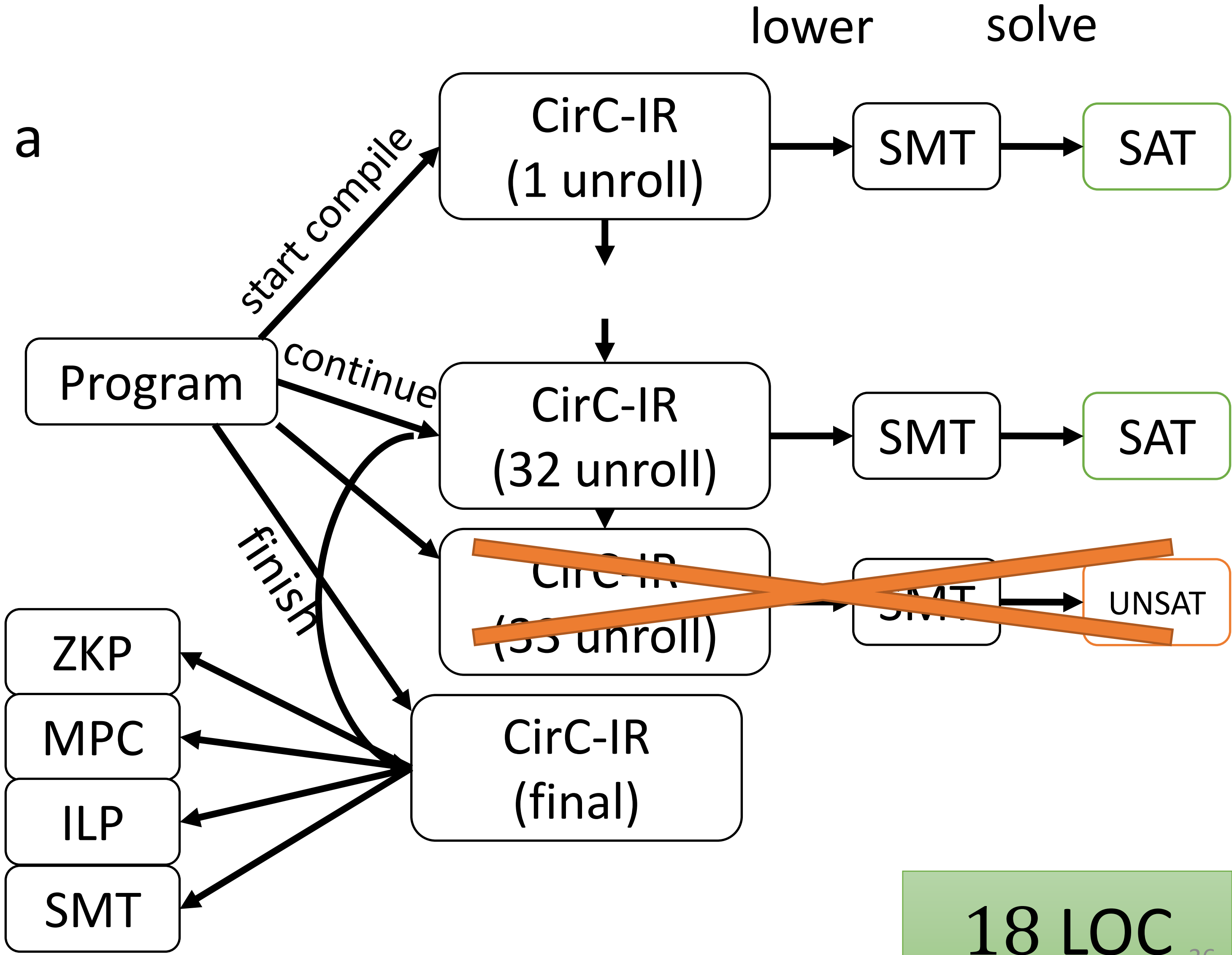
Supporting ZK Optimization Competitions



SMT-Assisted Optimization

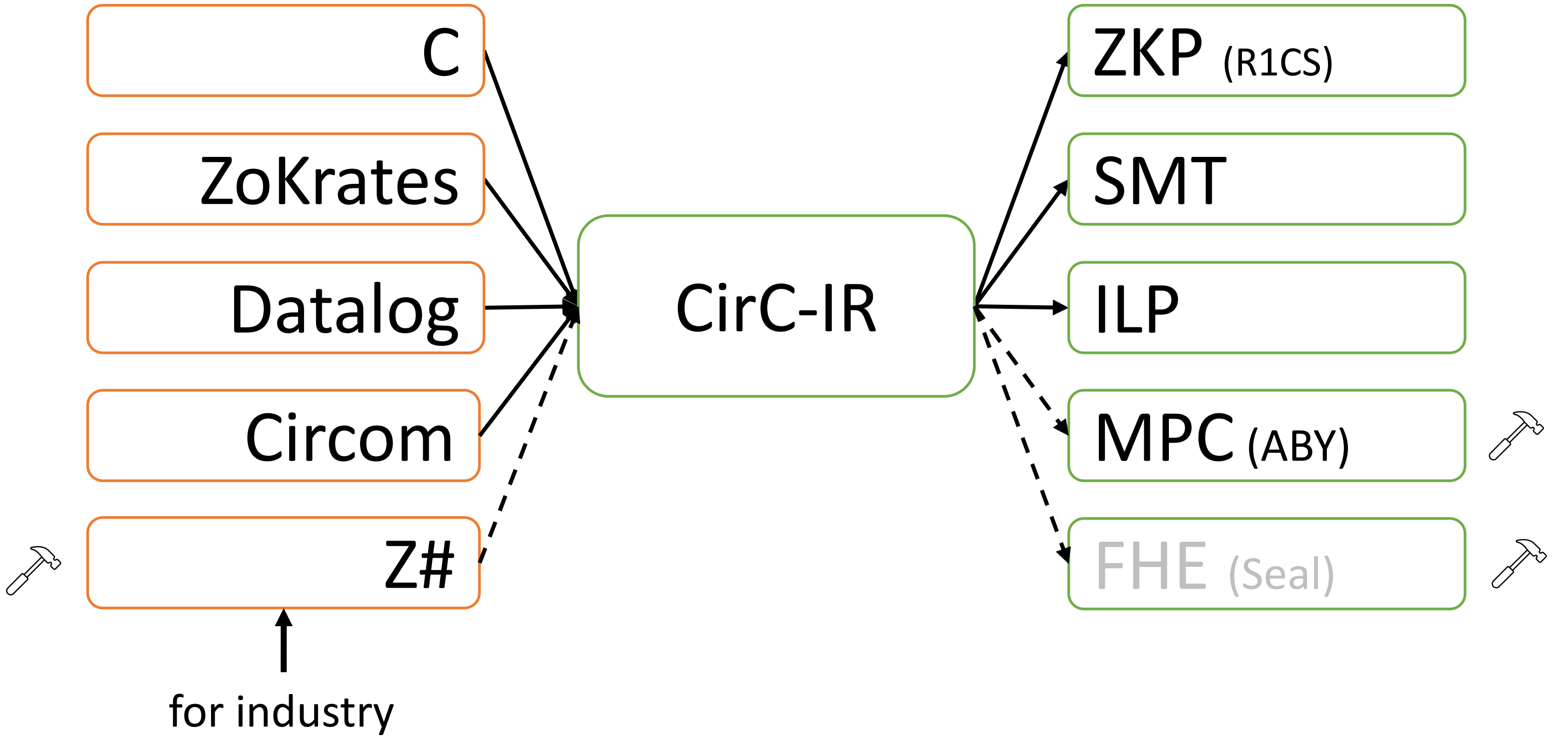
How many times do we unroll a loop?

- Too many → inefficient
- Too few → incomplete



18 LOC

Ongoing & Future Work

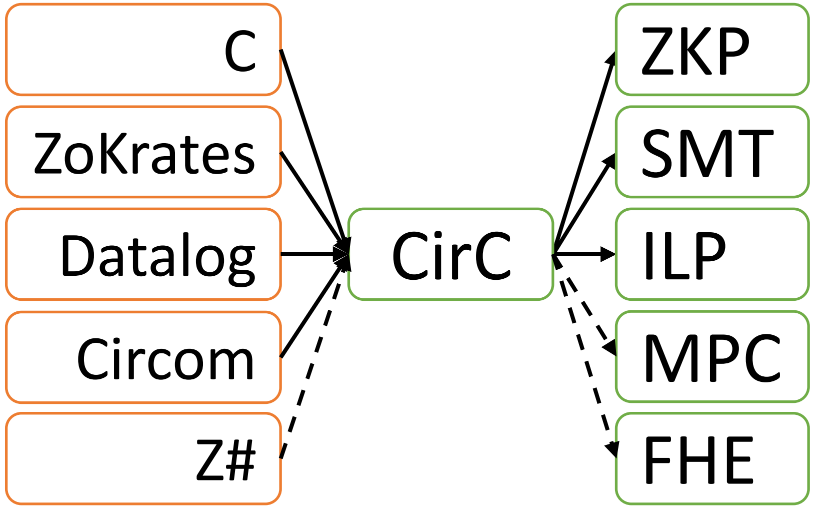


- Optimize for "stacked garbling" MPCs?
- Batching for MPC protocols?
- Compiling to "layered circuits"?

Builds on (Haskell) CirC: "Efficient Representation of Numerical Optimization Problems for SNARKs" https://ia.cr/2021/1436
Sebastian Angel and Andrew J. Blumberg and Eleftherios Ioannidis and Jess Woods, USENIX Security'22

CirC: Circuit Compiler Infrastructure

Different circuits can ***share*** compiler infrastructure



Based on ***EQCs***

(existentially quantified circuits)

Benefits:

- easy extension
- shared optimizations

Future directions:

- new front-ends
- new backends (and opportunities)
- ? *come talk to us!*

https://github.com/circify/circ

https://circ.zk.fyi