

# Writing Programs for ZKPs

Presented at NIST STPPA#8, 2025-Sep-18, Gaithersburg  
Special Topics on Privacy and Public Auditability (Event 8)

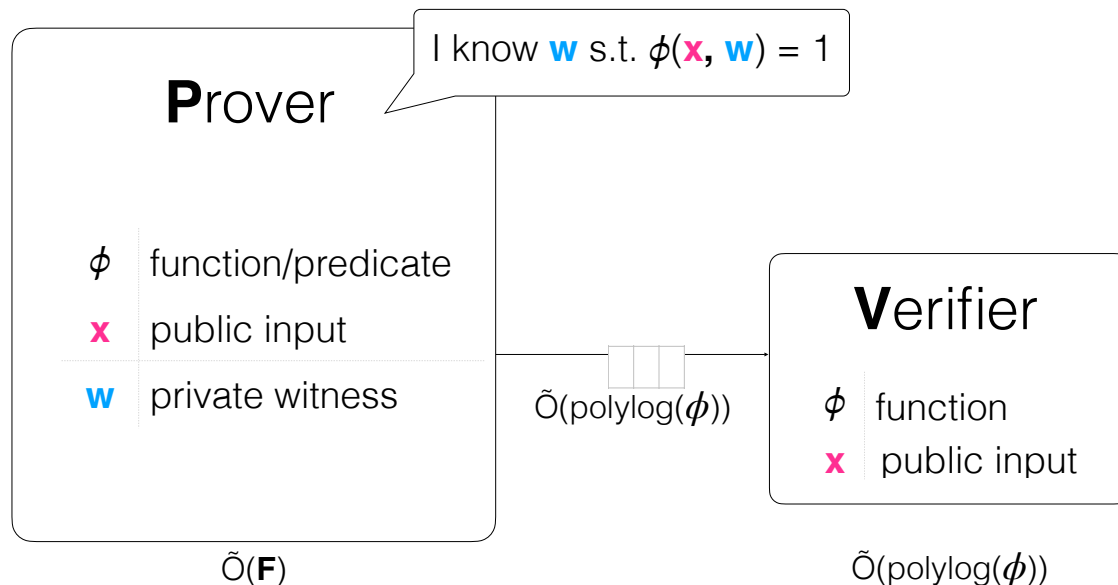


Pratyush Mishra



Alex Ozdemir

# Succinct Zero Knowledge Proofs [GMR85]



**Completeness:** For *true* theorems,  $\exists$  a prover that convinces verifier wp 1.

**Soundness:** For *false* theorems, no PPT prover can convince verifier w/ non-negl prob.

**Succinctness:** The verifier *runs in time*  $\text{polylog} |\phi|$ .

**Zero-knowledge:** The verifier learns nothing about  $w$  except that it's valid.

# Many applications!



**Private  
transactions**



**Aleo**



**celo**



**polygon**



**zkSync**



**AZTEC**

**MINA**



**STARKWARE**

...

**Scalable and/or Private  
Smart Contracts**



**Decentralized multiplayer  
games**

- **Anonymous credentials** [DFKP16]
- **Prove existence of security vulnerability** [DARPA Sieve, OBW22]
- **Coercion-resistant voting** [MACI]
- **Parsing of private documents**
- **Proof of ML training/inference**
- ...

# Using a ZKP



Idea →

Programmer

???

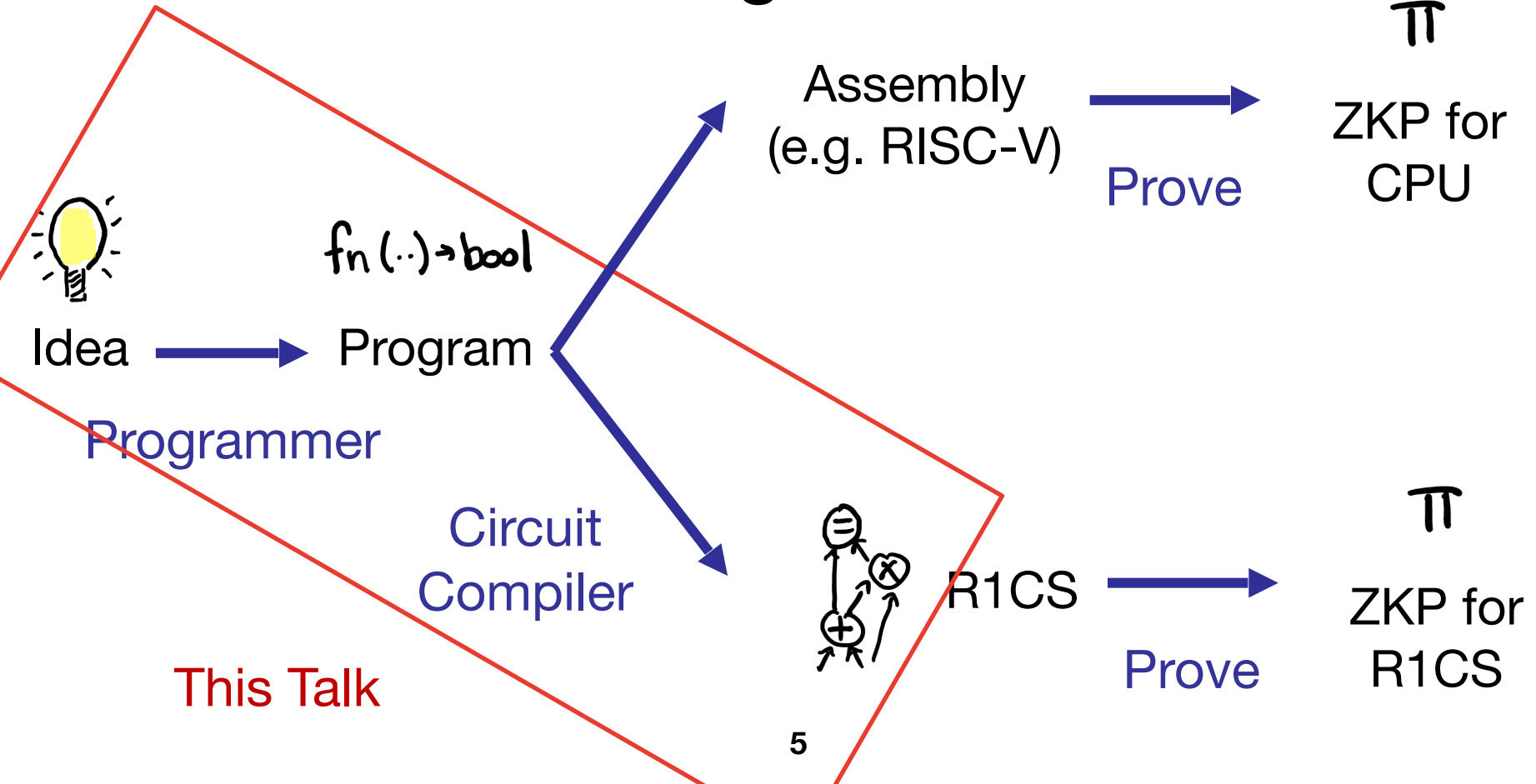
→

Prove

$\pi$

ZKP

# Using a ZKP



# This Talk

1. Big Picture: ZKP circuit programmability
2. Using an HDL (+ tutorial)
3. Using a library (+ tutorial)
4. Using a compiler (+ tutorial)
5. An overview of prominent ZKP toolchains

# ZKP Programmability

# Recap: ZKPs for a predicate $\phi$

- Prover knows  $\phi, x, w$
- Verifier knows  $\phi, x$
- Proof  $\pi$  shows that  $\phi(x, w)$  holds
  - but does not reveal  $w$
- Key Question: what can  $\phi$  be?

# What is $\phi$ ?

## $\phi$ in theory

- $w$  is a factorization of integer  $x$
- $w$  is the secret key for public key  $x$
- $w$  is the credential for account  $x$
- $w$  is a valid transaction

## $\phi$ in practice

- $\phi$  is an “arithmetic circuit” over inputs  $x, w$

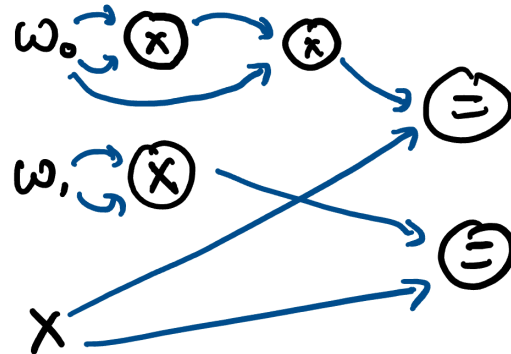
# Arithmetic Circuits (ACs), Part I

- Domain: “prime field”
  - $p$ : a large ( $\sim 255$  bit) prime
  - $\mathbb{Z}_p$ : the integers, mod  $p$ 
    - operations:  $+$ ,  $\times$ ,  $=$  (mod  $p$ )
  - Example in  $\mathbb{Z}_5$ :
    - $4 + 5 = 9 = 4$
    - $4 \times 4 = 16 = 1$
- ACs as systems of field equations:
  - Example:
    - $w_0 \times w_0 \times w_0 = x$
    - $w_1 \times w_1 = x$
    - Addition is also OK

# Arithmetic Circuits (ACs), Part II

- ACs as circuits
  - Directed, acyclic graph
  - Nodes: inputs, gates, constants
  - Edges: wires/connections

- Example:
  - $w_0 \times w_0 \times w_0 = x$
  - $w_1 \times w_1 = x$
- As a circuit:

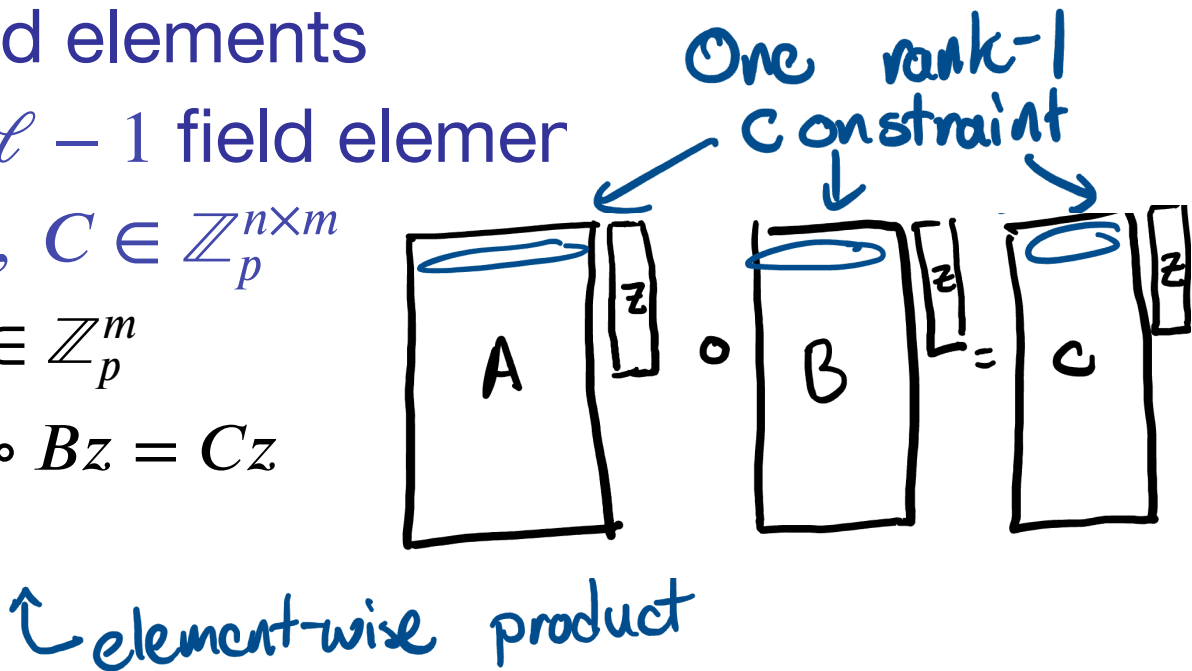


# R1CS: a common Arithmetic Circuit format

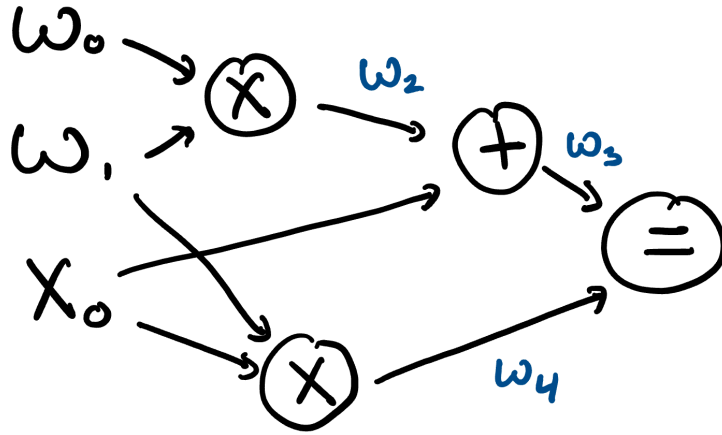
- R1CS: format for ZKP ACs
- Definition:
  - $x$ : field elements  $x_1, \dots, x_\ell$
  - $w$ :  $w_1, \dots, w_{m-\ell-1}$
  - $\phi$ :  $n$  equations of form
    - $\alpha \times \beta = \gamma$
    - where  $\alpha, \beta, \gamma$  are affine combinations of variables
- Examples:
  - $w_2 \times (w_3 - w_2 - 1) = x_1$
  - $w_2 \times w_2 = w_2$
  - ~~■  $w_2 \times w_2 \times w_2 = x_1$~~ 
    - $w_2 \times w_2 = w_4$
    - $w_4 \times w_2 = x_1$

# R1CS: Matrix Definition

- $x$ : vector of  $\ell$  field elements
- $w$ : vector of  $m - \ell - 1$  field elements
- $\phi$ : matrices  $A, B, C \in \mathbb{Z}_p^{n \times m}$ 
  - $z = (1 \parallel x \parallel w) \in \mathbb{Z}_p^m$
  - Holds when  $Az \circ Bz = Cz$

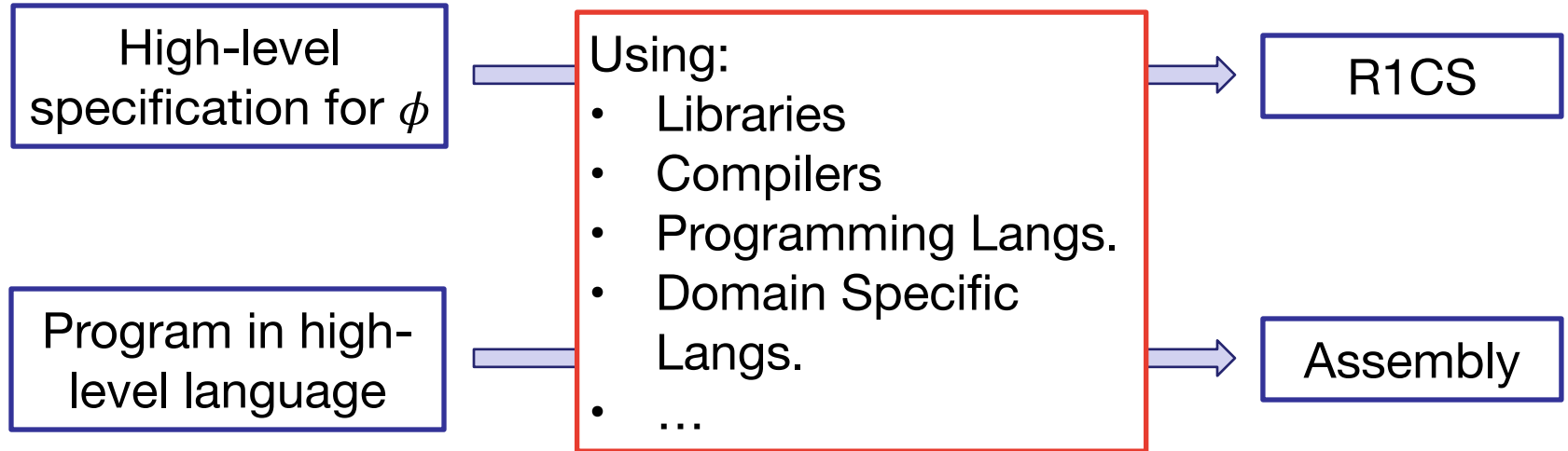


# Writing an AC as R1CS (Example)



- Step 1: intermediate  $w$ 's
- Step 2: write equations
  - $w_0 \times w_1 = w_2$
  - $w_3 = w_2 + w_4$
  - $w_1 \times x_0 = w_4$
  - $w_3 = w_4$

# Zooming out: a Programming Languages problem



# The Idea



- Booleans
- Structures
- Modules
- Functions
- ...

# An Example



- Merkle tree
- Pedersen Hash
- Signatures
- Spend logic
- ...

# An HDL for R1CS

# Programming Languages (PLs) vs. Hardware Description Languages (HDLs)

## PL objects

- Variables
- Operations
- Program/Functions

## PL actions

- Mutate variables
- Call functions

## HDL objects

- Wires
- Gates
- Circuit/Sub-circuits

## HDL actions

- Connect wires
- Create sub-circuits

# HDLs: From Digital to Arithmetic

## HDLs for Digital Circuits

- Verilog
- SystemVerilog
- VHDL
- Chisel
- ...

## An HDL for R1CS

- circom
  - wires: R1CS variables
  - gates: R1CS constraints
- a circom circuit does 2 things:
  - sets variable values
  - creates R1CS constraints

# Circom: Base Language

- A “template” is a (sub)circuit
- A “signal” is a wire
  - “input” or “output”
- “<--” sets signal values
- “===” creates constraints
  - Must be rank-1:
    - one side: linear
    - other side: quadratic
- “<==” does both

```
template Multiply() {
    signal input x;
    signal input y;
    signal output z;

    z <-- x * y;
    z === x * y;
    // ERROR: z === x * x * y
    // OR    : z <== x * y;
}

component main {public [x]} =
    Multiply();
```

# Circom: Metaprogramming Language

- Template arguments
- Signal arrays
- Variables
  - Mutable
  - Not signals
  - Evaluated at compile-time
- Loops
- If statements
- Array accesses

```
template RepeatedSquaring(n) {
    signal input x;
    signal output y;

    signal xs[n];
    xs[0] <== x;
    for (var i = 0; i <= n; i++) {
        xs[i+1] <== xs[i] * xs[i];
    }
    y <== xs[n];
}

component main {public [x]} =
    RepeatedSquaring(1000);
```

# Circom: Witness Computation & Sub-circuits

- Witness computation: more general than R1CS
  - “<--” is more general than “===”
- “component”s hold sub-circuits
  - Access inputs/outputs with dot-notation

```
template NonZero() {  
    signal input in;  
    signal inverse;  
    inverse <-- 1 / in; // not R1CS  
    1 == in * signal; // is R1CS  
}
```

```
template Main() {  
    signal input a; signal input b;  
    component nz = NonZero();  
    nz.in <== a;  
    0 == a * b;  
}
```

# Circom Code Example

# Tutorial Example: Sudoku

- 9 by 9 grid
- Some cells have #s
- Goal: fill all cells with 1...9
- Rule: no duplicates in any:
  - Column
  - Row
  - 3x3 sub-grid



7	5		9					6
	2	3		8				4
8					3			1
5			7	2				
	4		8	6			2	
			9	1				3
9			4					7
	6			7		5	8	
7			1		3	9		

1	7	5	2	9	4	8	3	6
6	2	3	1	8	7	9	4	5
8	9	4	5	6	3	2	7	1
5	1	9	7	3	2	4	6	8
3	4	7	8	5	6	1	2	9
2	8	6	9	4	1	7	5	3
9	3	8	4	2	5	6	1	7
4	6	1	3	7	9	5	8	2
7	5	2	6	1	8	3	9	4

Puzzle



Solution



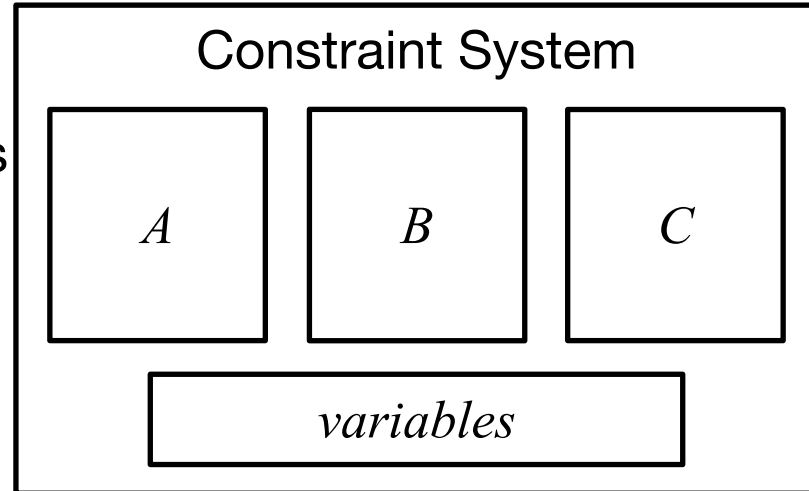
# A Library for R1CS

# Circom: Recap

- An HDL for R1CS
- Key features:
  - Direct control over constraints
  - Custom language
    - Can be good
    - Can be bad

# R1CS Libraries

- A library in a *host* language (Eg: Rust, OCaml, C++, Go, ...)
- Key type: *constraint system*
  - Maintains state about R1CS constraints and variables
- Key operations:
  - create variable
  - create *linear combinations* of variables
  - add constraint



# ConstraintSystem Operations

## Variable creation:

`cs.add_var(p, v) → id`

- `cs`: constraint system
- `p`: visibility of variable
- `v`: assigned value
- `id`: variable handle

## Linear Combination creation:

`cs.zero()` : returns the empty LC

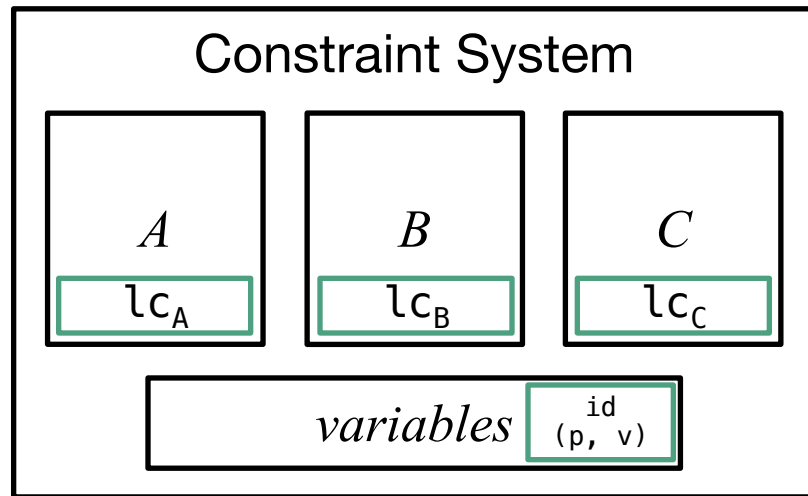
`lc.add(c, id) → lc'`

- `id`: variable
- `c`: coefficient
- `lc' := lc + c * id`

## Adding constraints:

`cs.constrain(lcA, lcB, lcC)`

- Adds a constraint  $lc_A \times lc_B = lc_C$



# Example: Boolean AND

1. Create result variable

```
fn and(cs: ConstraintSystem, a: Var, b: Var) → Var {  
  let result = cs.new_witness_var(|| a.value() & b.value());  
  self.cs.enforce_constraint(  
    lc!() + a,  
    lc!() + b,  
    lc!() + result,  
  );  
  result  
}
```

3. Enforce constraint

2. Create linear combinations

# Example: Boolean AND

Create result variable

```
fn and(cs:  
    let res  
    self.cs  
    lc  
    lc  
    lc  
);  
result  
}
```

This is unpleasant, tedious, and error-prone!

Can you imagine writing a complex algorithm like signature verification in this style?

# Idea: Leverage Language Abstractions!

We can use language abstractions like structs, operator overloading, methods, etc. to allow better developer UX:

Wrap variable in dedicated type

```
struct Boolean { var: Var };  
  
impl BitAnd for Boolean {  
    fn and(self: Boolean, other: Boolean) → Boolean {  
        // Same as before  
        Boolean { var: result }  
    }  
}
```

Implement interface for operator overloading

# Does it work? Yes!

Can use abstractions like normal code:

```
let a = Boolean::new_witness(|| true);  
let b = Boolean::new_witness(|| false);  
(a & b).enforce_equal(Boolean::FALSE);
```

Many different gadget libraries:

- libsnark: gadgetlib (C++)
- arkworks: r1cs-std + crypto-primitives (Rust)
- Bellman (Rust)
- Snarky (Ocaml)
- Gnark (Go)

# What about Witness Computation?

- Can perform arbitrary computations to generate witnesses

Closure (lambda) executed only during proving!

```
let a = Boolean::new_witness(|| (4 == 5) & (x < y));  
let b = Boolean::new_witness(|| false);  
(a & b).enforce_equal(Boolean::FALSE);
```

# Arkworks Code Example

# Compiling a Programming Language to R1CS

# HDLs & Circuit Libraries

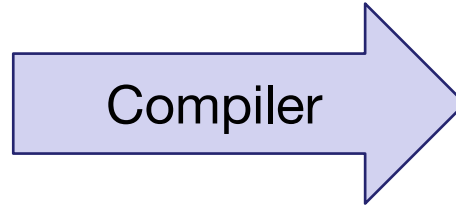
- Difference:
  - Host language v. custom language
- Similarities:
  - explicit wire creation (explicitly wire values)
  - explicit constraint creation
- Do we need to explicitly build a circuit?
  - No!

# Compiling PLs to Circuits (Idea)

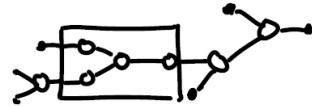
Program

```
fn main(...) {  
  ...  
}
```

- Variables
- Mutation
- Functions
- Arrays



R1CS



- Wires
- Constraints

# ZoKrates: Syntax

- Struct syntax for custom types
- Variables contain values during execution/proving
- Can annotate privacy
- “assert” creates constraints

```
type F = field;
```

```
def main(public F x, private F[2] ys)  
{  
    field y0 = y[0];  
    field y1 = y[1];  
    assert(x = y0 * y1);  
}
```

# Zokrates: Language features

- Integer generics

- Arrays

- Variables

  - Mutable

- Fixed-length loops

- If expressions

- Array accesses

```
def repeated_squaring<N>(field x) ->
field {
    field[N] mut xs;
    xs[0] = x;
    for u32 i in 0..n {
        xs[i + 1] = xs[i] * xs[i];
    }
    return xs[N];
}
def main (public field x) -> field {
    repeated_squaring::<1000>(x)
}
```

# What about Witness Computation?

- No way to compute witnesses
- All witnesses must be provided as input

```
def main(private field a, public field b) {  
    assert(a * b == 1)  
}
```

# ZoKrates

# Examples

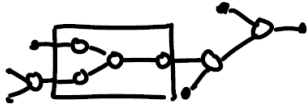
**ZKP**

**Toolchains:  
A Quick Tour**

# Toolchain Type

## HDL

*a language for  
describing circuit  
synthesis*



## Library

*a library for  
describing circuit  
synthesis*

```
circ.add_wire(...)  
circ.add_gate(...)
```

## PL + Compiler

*a language,  
compiled to a circuit*

```
fn main(...){  
  ...  
}
```

# Toolchain Types, Organized

		Standalone Language?	
		No	Yes
Language Type	Circuit	Library (arkworks)	HDL (circom)
	Program		PL (noir)

## circom

Pros:

- Clear constraints
- Elegant syntax

Cons:

- Hard to learn
- Limited abstraction

types?

===

HDL

## arkworks

Pros:

- Clear constraints
- As expressive as Rust

Cons:

- Need to know Rust
  - Few optimizations
- manual opts

## ZoKrates

Pros:

- Easiest to learn
- Elegant syntax

Cons:

- Limited witness computation

always implicit

just a PL

# List of toolchains

## HDL

- Circom

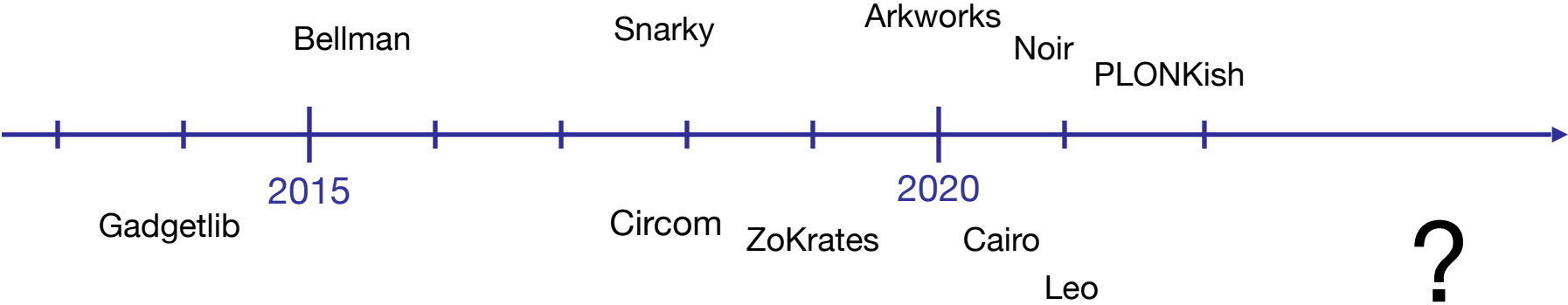
## Library

- Arkworks (Rust)
- Gadgetlib (C++)
- Bellman (Rust)
- Snarky (OCaml)
- PLONKish (Rust)

## PL + Compiler

- ZoKrates
- Noir
- Leo
- Cairo

# Timeline



# Shared Compiler Infrastructure?

## Source

Cairo  
ZoKrates  
Circom  
PLONKish  
Noir  
Snarky  
Bellman  
Gadgetlib  
Leo

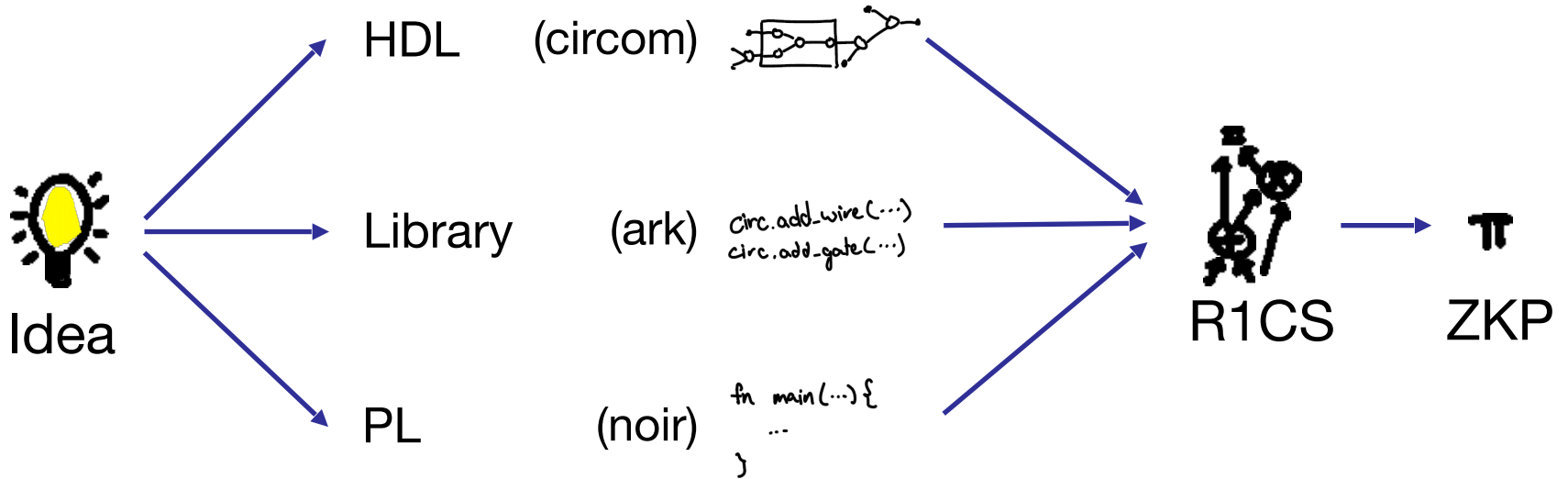
## Common Techniques



## Target

R1CS  
Plonk  
AIR

# Summary



# Thank you!

Code available at

<https://github.com/rdi-berkeley/zkp-course-lecture3-code>