

Leo.

**A Programming Language for Formally Verified,
Zero-Knowledge Applications**

Collin Chin Howard Wu

Raymond Chu Alessandro Coglio

Eric McCarthy Eric Smith



**Decentralized applications have become
a foundation for new paradigms.**

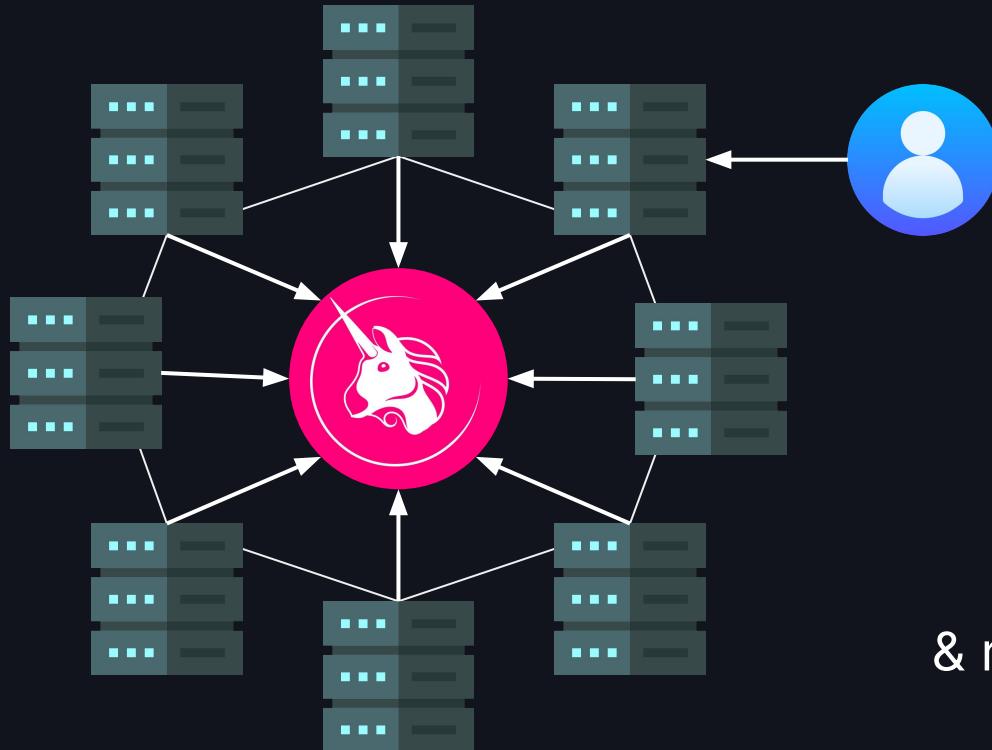




Decentralization has applications now become
**Yet this new paradigm
comes with significant challenges.**



The Scalability Problem



Wasteful

Each miner must re-execute
every transaction

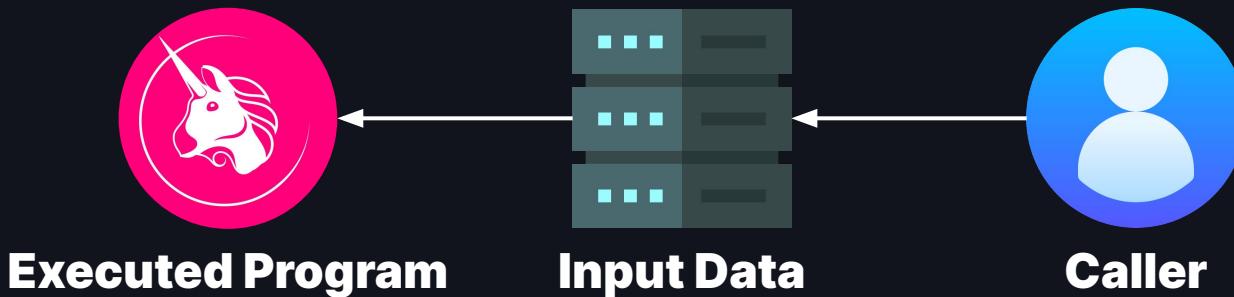
Constrained

Limited running time,
minimal stack size,
& restrictive instruction sets

The Privacy Problem

Their core strength is also their primary weakness:
the history of all state transitions must be executed by all parties.

Anyone Can See



Miner-Extractable Value (MEV)

Lack of privacy enables **miner frontrunning** and **arbitrage attacks**

The Auditability Problem

Decentralized applications offer
weak guarantees of correctness and safety.



Writing zero-knowledge applications requires
specialized domain expertise.

Leo.

A new programming language for
formally verified, zero-knowledge applications.

Composability Users can combine arbitrary functions

Decidability Users know what the compiled program executes

Universality Users don't need to run their own ceremony

With Leo, applications are:

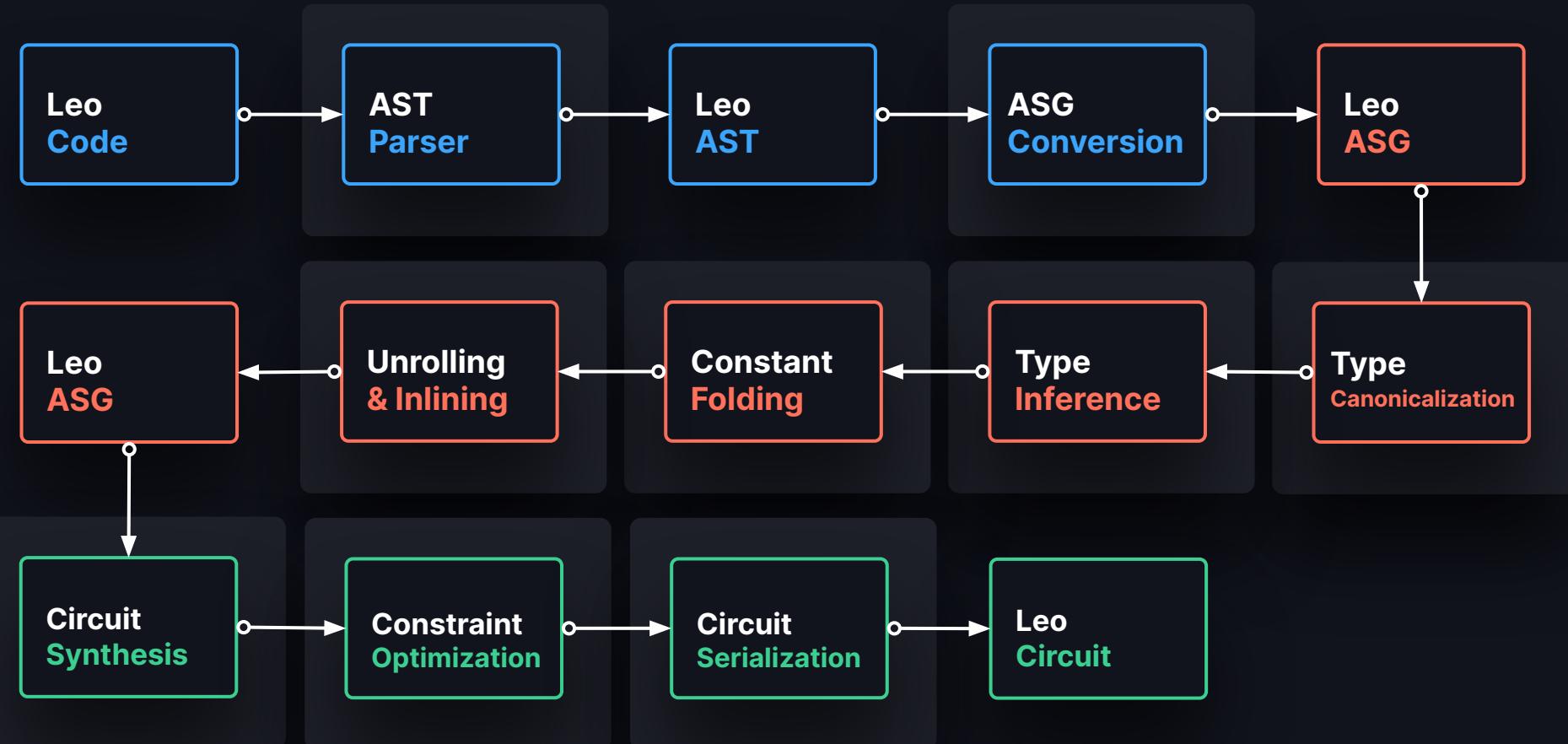
Verifiably Computed

Executions can be succinctly verified by anyone

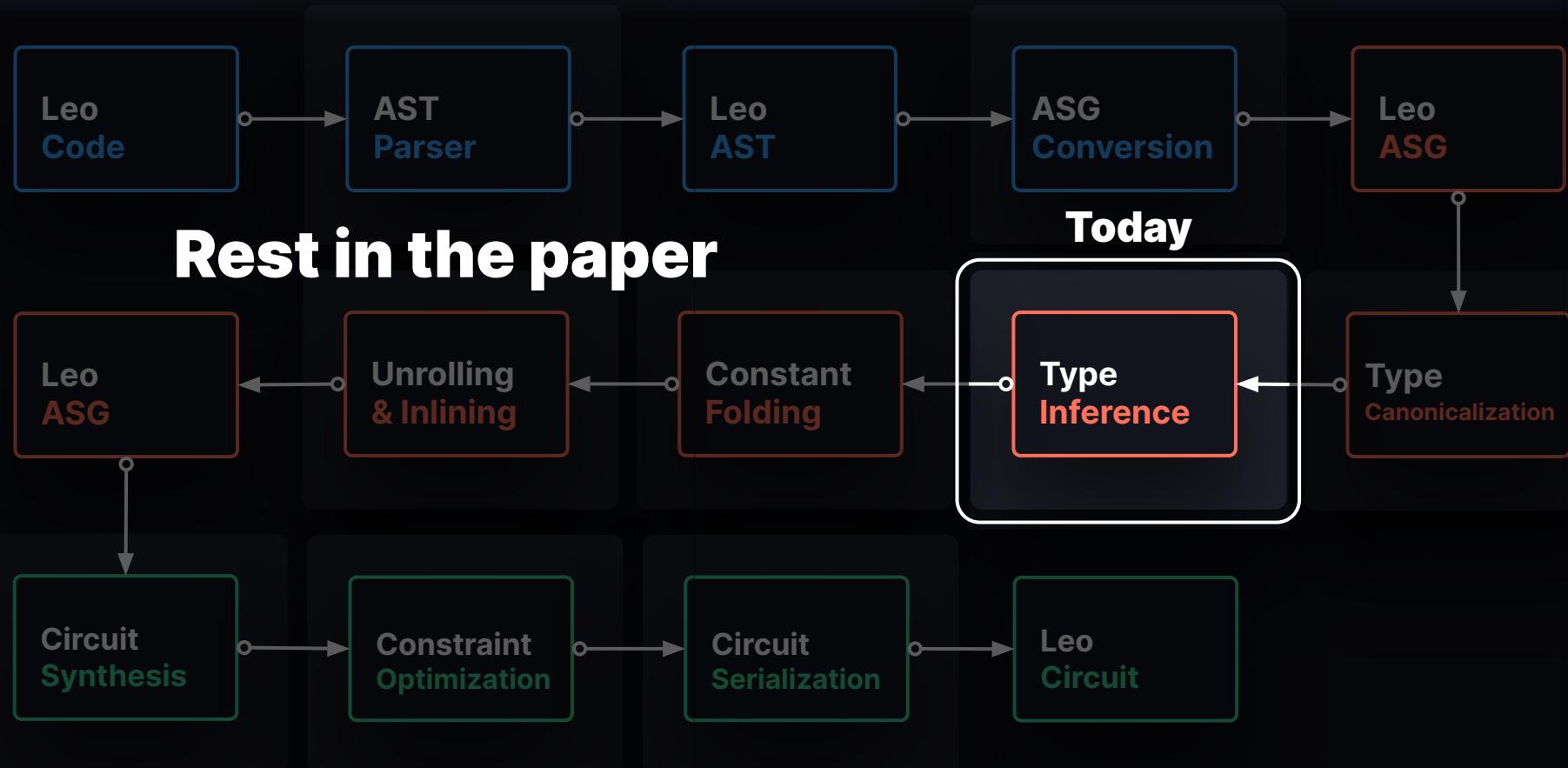
Correct-by-Construction

Compiled programs are formally verified

The Leo Compiler



The Leo Compiler



Type Inference

```
function foo() {  
    let a = 3u8;  
    let b: i16 = -5;  
    let c = 100;  
    let d = 100;  
    let e = [c, 8i8];  
    let f = bar(d);  
}  
  
function bar(g: field) -> bool {  
    ...  
}
```

Type Inference

```
function foo() {  
    let a: u8 = 3u8;  
    let b: i16 = -5i16;  
    let c: i8 = 100i8;  
    let d: field = 100field;  
    let e: [i8; 2] = [c, 8i8];  
    let f: bool = bar(d);  
}  
  
function bar(g: field) -> bool {  
    ...  
}
```

Automatic detection of the type of an expression in a formal language.

Type Inference

```
function foo() {  
    let a = 3u8;  
    let b: i16 = -5;  
    let c = 100;  
    let d = 100;  
    let e = [c, 8i8];  
    let f = bar(d);  
}  
  
function bar(g: field) -> bool {  
    ...  
}
```

Assign
Variables

Automatic detection of the type of an expression in a formal language.

Step One

```
function foo() {  
    let a:  $T_1$  = 3u8;  
    let b: i16 = -5 $T_2$ ;  
    let c:  $T_3$  = 100 $T_4$ ;  
    let d:  $T_5$  = 100 $T_6$ ;  
    let e:  $T_7$  = [c, 8i8];  
    let f:  $T_8$  = bar(d);  
}  
  
function bar(g: field) -> bool {  
    ...  
}
```

Type Inference

Automatic detection of the type of an expression in a formal language.

Step One

```
function foo() {  
    let a:  $T_1$  = 3u8; ...  
    let b: i16 = -5 $T_2$ ; ...  
    let c:  $T_3$  = 100 $T_4$ ; ...  
    let d:  $T_5$  = 100 $T_6$ ; ...  
    let e:  $T_7$  = [c, 8i8]; ...  
    let f:  $T_8$  = bar(d); ...  
}  
  
function bar(g: field) -> bool {  
    ...  
}
```

Step Two

$T_1 \equiv u8$
 $i16 \equiv T_2$
 $T_3 \equiv T_4$
 $T_5 \equiv T_6$
 $T_7 \equiv [T_3; 2]$
 $T_5 \equiv \text{field}$
 $T_8 \equiv \text{bool}$

Step Three

$T_1 = u8$
 $T_2 = i16$
 $T_3 = i8$
 $T_4 = i8$
 $T_5 = \text{field}$
 $T_6 = \text{field}$
 $T_7 = [i8; 2]$
 $T_8 = \text{bool}$

Form
Equations

Infer
Types

Type
Inference

Automatic detection of the type of an expression in a formal language.

Type Inference Issues

- Compilers are known to be buggy
- Inferring an incorrect type poses serious consequences
 - Incorrect output value → Loss of funds
 - Incorrect gadget selection → Broken protocol

How can we verify type inference is implemented correctly?

Type
Inference

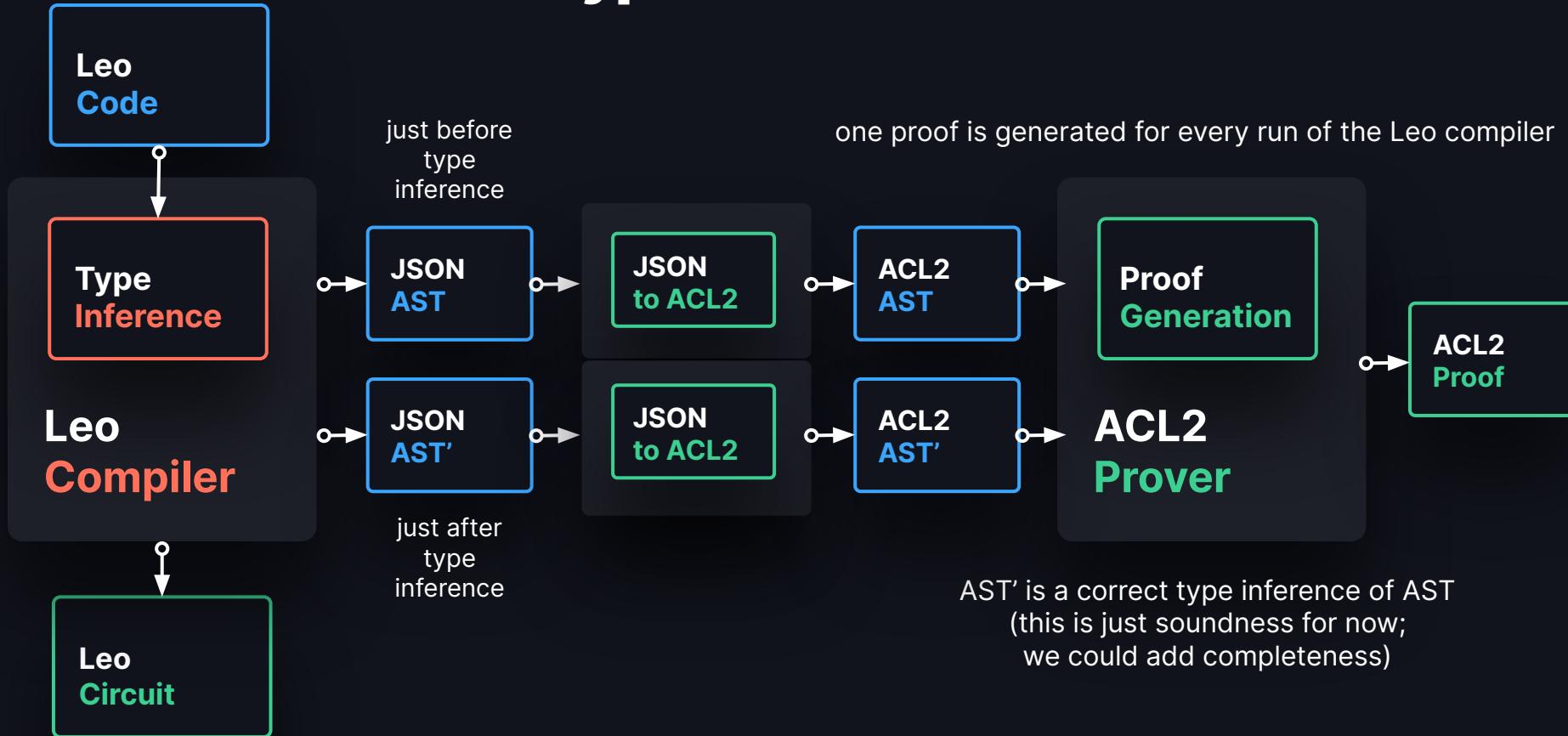
**Automatic detection of the type of an expression
in a formal language.**

forall
exists
not
implies

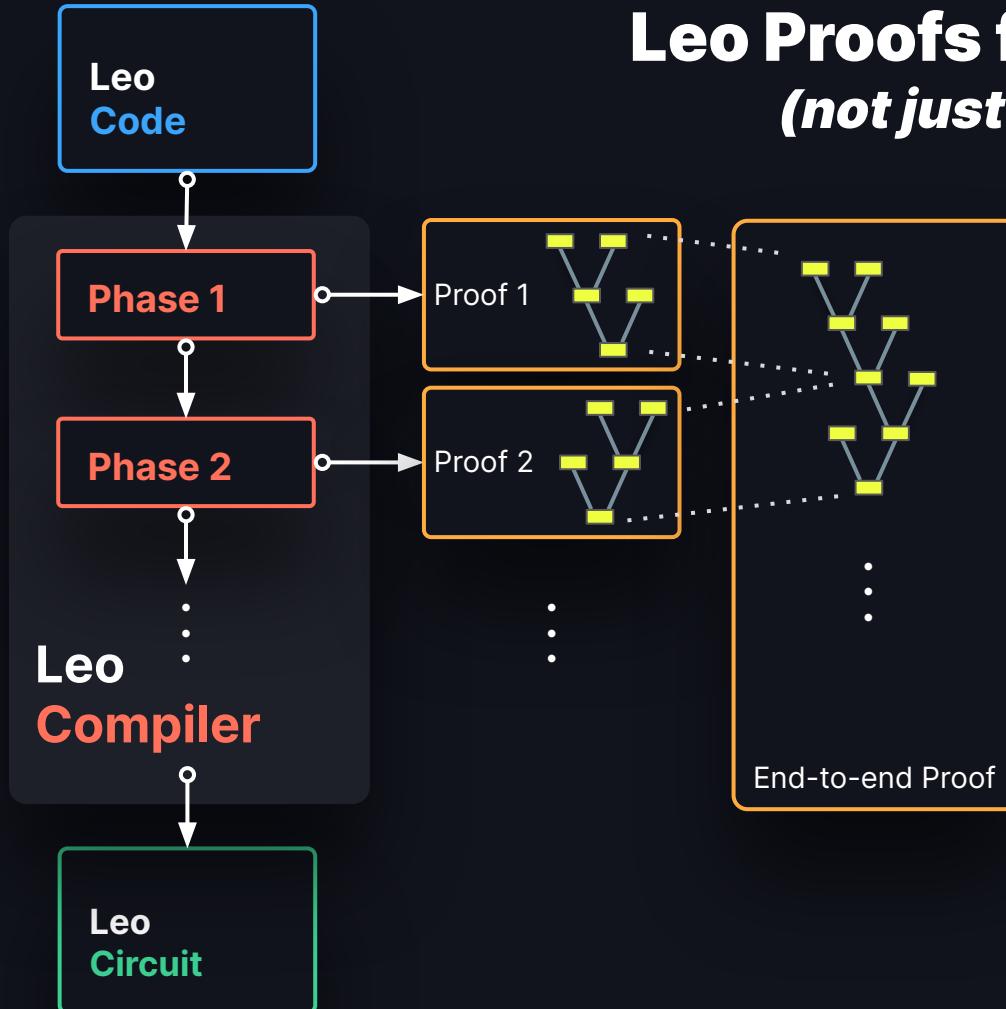
Formal Methods

- Formal methods are techniques and tools based on formal logic that aim to establish desired properties of programs by way of mathematical proofs.
- We use the **industrial-strength ACL2 theorem prover** in Leo.
- Leo is a **verifying compiler** - one that generates, every time it is run, a proof of the semantic equivalence between its input and output.

Leo Type Inference Proof



Leo Proofs for All the Phases *(not just Type Inference)*



- An end-to-end proof for every compiler run.
- Modular design allows for composable proofs.

Our Implementation of Leo



The Leo compiler is proudly written in Rust.

Open-source, 12 contributors, 65,000+ LoC



Leo programs are formally verified in ACL2.

Open-source (soon), partial passes complete

Leo.

Leo language formal specifications are public.

Open-source, formal definitions, 70+ pages

Performance of the Leo Compiler

	256-bit Pedersen Hash	Bubble Sorting Algorithm	Least-Squares Linear Regression
Time from code to grammar	0.160 ms	0.194 ms	0.504 ms
Time from grammar to AST	0.154 ms	0.204 ms	5.659 ms
Time from AST to ASG	0.084 ms	0.079 ms	0.245 ms
Time from ASG to R1CS	24.882 ms	70.393 ms	1,651.012 ms
Total compile time	25.280 ms	70.871 ms	1,657.420 ms
Number of constraints	1,539 gates	16,115 gates	433,867 gates

LEO CODE

```
1 // The 'joyful-jumpy-jubilant-jellyfish-pedersen-hash' main function.
2 circuit PedersenHash {
3     parameters: [group; 256];
4     // Instantiates a Pedersen hash circuit.
5     function new(const parameters: [group; 256]) → Self {
6         return Self { parameters };
7     }
8     // Returns the Pedersen hash for given message.
9     function hash(self, message: [bool; 256]) → group {
10        let digest: group = 0;
11        for i in 0..256 {
12            if message[i] {
13                digest += self.parameters[i];
14            }
15        }
16        return digest;
17    }
18 }
19
20 function main(hash_input: [bool; 256], const parameters: [group; 256]) → group {
21     const pedersen = PedersenHash::new(parameters);
22     return pedersen.hash(hash_input);
23 }
```

INPUTS

```
1 // The program input for joyful-jumpy-jubilant-jellyfish/src/main.leo
2 [main]
3 hash_input: [bool; 256] = [true; 256];
4
5 [constants]
6 parameters: [group; 256] = [igroup; 256];
7
8 [registers]
9 r0: group = (1, 0)group;
```

OUTPUTS

- Proving Key
- Verifying Key
- Proof
- main.output

STDOUT / STDERR

```
Build Starting...
Build Compiling main program... ("~/usr/playground-projects/joyful-jumpy-jubilant-jellyfish/src/main.leo")
Build Number of constraints - 1539

Build Complete
Done Finished in 39 milliseconds

Setup Starting...
Setup Saving proving key ("~/usr/playground-projects/joyful-jumpy-jubilant-jellyfish.lpk")
Setup Complete
Setup Saving verification key ("~/usr/playground-projects/joyful-jumpy-jubilant-jellyfish/outputs/joyful-jumpy-jubilant-jellyfish.lv")
Setup Complete
Done Finished in 207 milliseconds

Proving Starting...
Proving Saving proof... ("~/usr/playground-projects/joyful-jumpy-jubilant-jellyfish/outputs/joyful-jumpy-jubilant-jellyfish.proof")
Done Finished in 166 milliseconds
```

The interface is a dark-themed web application for Aleo Studio. It features a central workspace divided into sections: LEO CODE, INPUTS, OUTPUTS, and STDOUT / STDERR. The LEO CODE section contains the source code for a Pedersen hash circuit and its main function. The INPUTS section shows the program's inputs. The OUTPUTS section lists the generated artifacts: Proving Key, Verifying Key, Proof, and main.output. The STDOUT / STDERR section displays the build process, which includes compilation, setup of proving and verification keys, and saving of the proof. On the right side, there is a vertical toolbar with icons for play, help, and other studio functions. At the bottom, there are social sharing icons and a "Powered by Aleo" logo.

The World's First IDE for Zero-Knowledge Proofs.

Welcome to Aleo Studio.

Imagine a world of truly private applications.

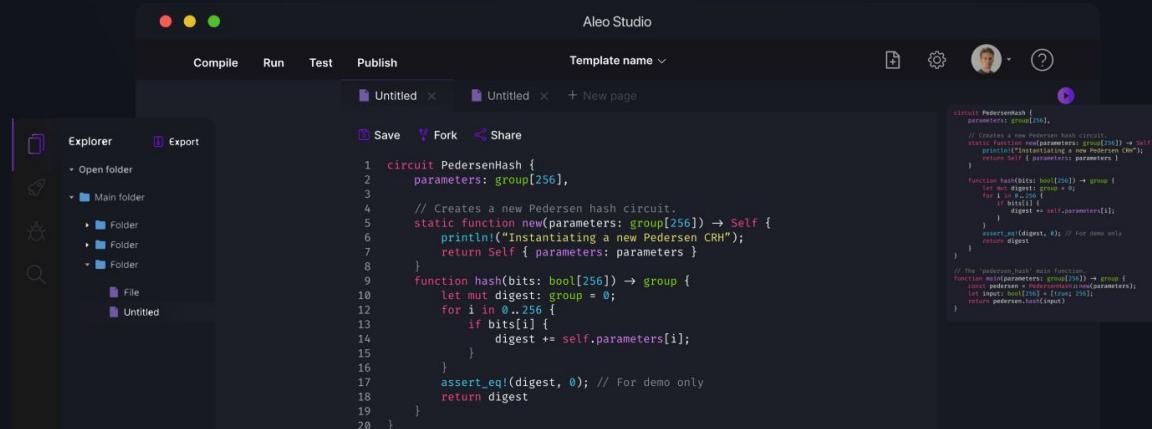
Windows

Linux

macOS

Download Alpha

macOS 10.13.0 and newer



Aleo Studio

Compile Run Test Publish Template name ▾

Untitled × Untitled × + New page

Explorer Export

Open folder Main folder Folder File Untitled

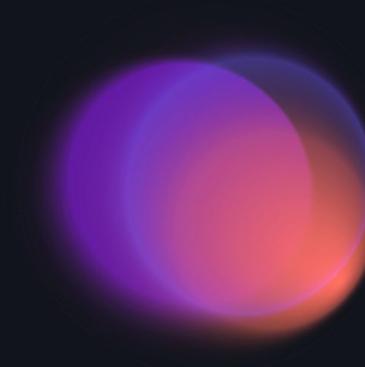
```
circuit PedersenHash {
    parameters: group[256],
    // Creates a new Pedersen hash circuit.
    static function new(parameters: group[256]) → Self {
        println!("Instantiating a new Pedersen CRH");
        return Self { parameters: parameters };
    }
    function hash(bits: bool[256]) → group {
        let mut digest: group = 0;
        for i in 0..256 {
            if bits[i] {
                digest += self.parameters[i];
            }
        }
        assert_eq!(digest, 0); // For demo only
        return digest;
    }
} // The 'pedersen_hash' main function.
```

<https://aleo.studio>

Where Applications Become Private

Aleo Package Manager

Become an Aleo developer and contribute to the first registry for zero-knowledge circuits.



 Search for packages

Search

Most Downloaded

New Packages

Just Updated

howard/silly-sudoku



 0.1.2  MIT  1126

A simple Sudoku puzzle grid

 Public

 5.3 KB

leobot/test-app



 0.1.600235820  862

argus4130/xnor



 0.1.0  LICENSE-MIT  750

The xnor package

 Public

 8.3 KB

<https://aleo.pm>

Conclusion

Leo.



Leo Programming Language
<https://leo-lang.org>



Full Paper is Available Online
<https://paper.leo-lang.org>

Leo Github
<https://github.com/AleoHQ/leo>