Leo.

A Programming Language for Formally Verified, Zero-Knowledge Applications

Collin Chin Howard Wu
Raymond Chu Alessandro Coglio
Eric McCarthy Eric Smith

ZKPROOF WORKSHOP, 2021




Decentralized applications have become a foundation for new paradigms.





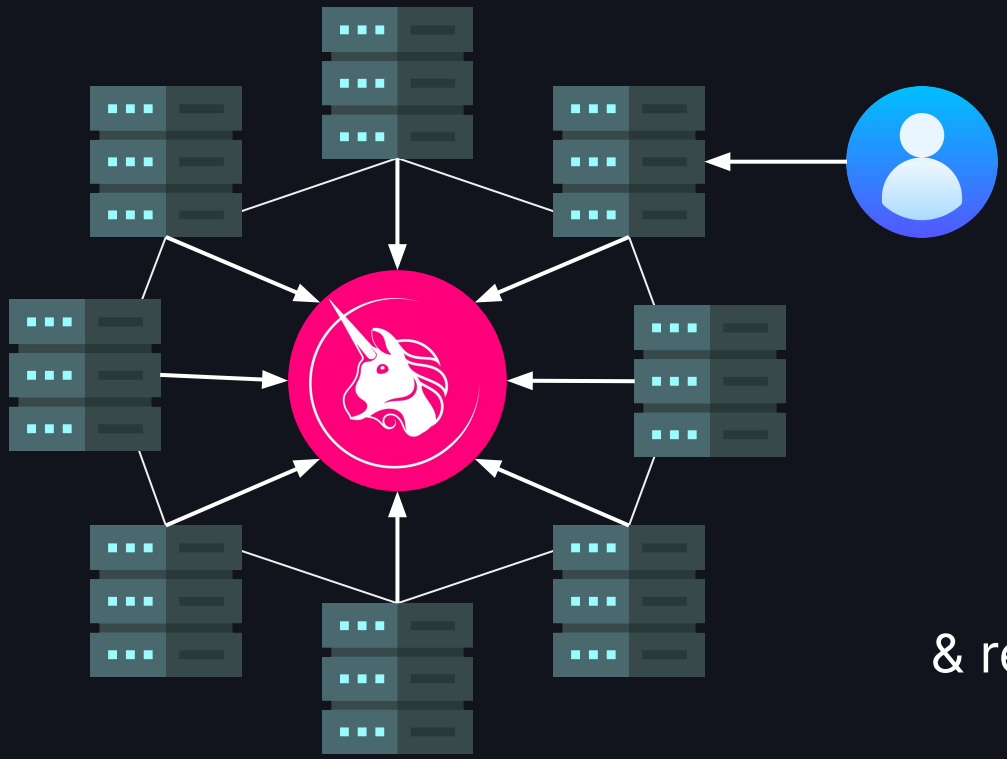



Yet this new paradigm comes with significant challenges.





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.

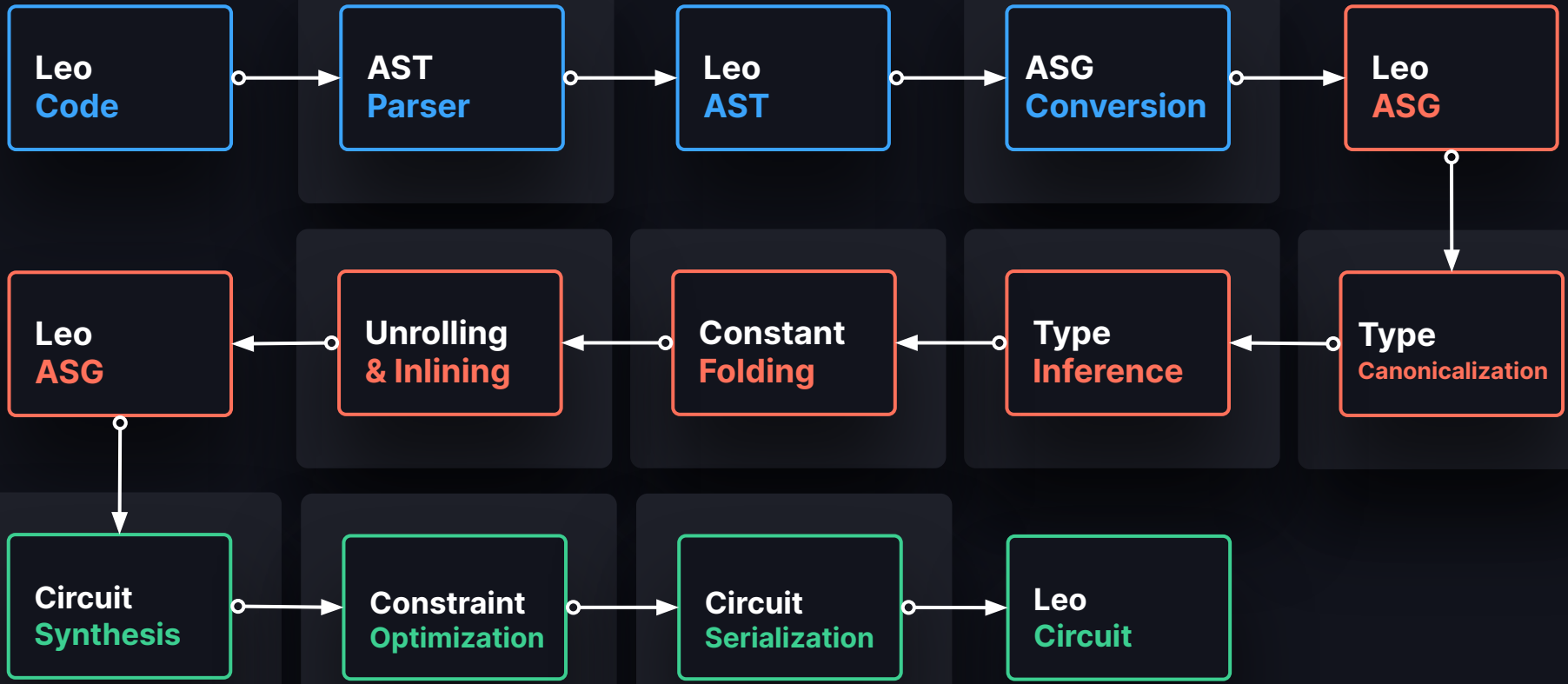
Composability	Users can combine arbitrary functions
Decidability	Users know what the compiled program executes
Universality	Users don’t need to run their own ceremony

With Leo, applications are:

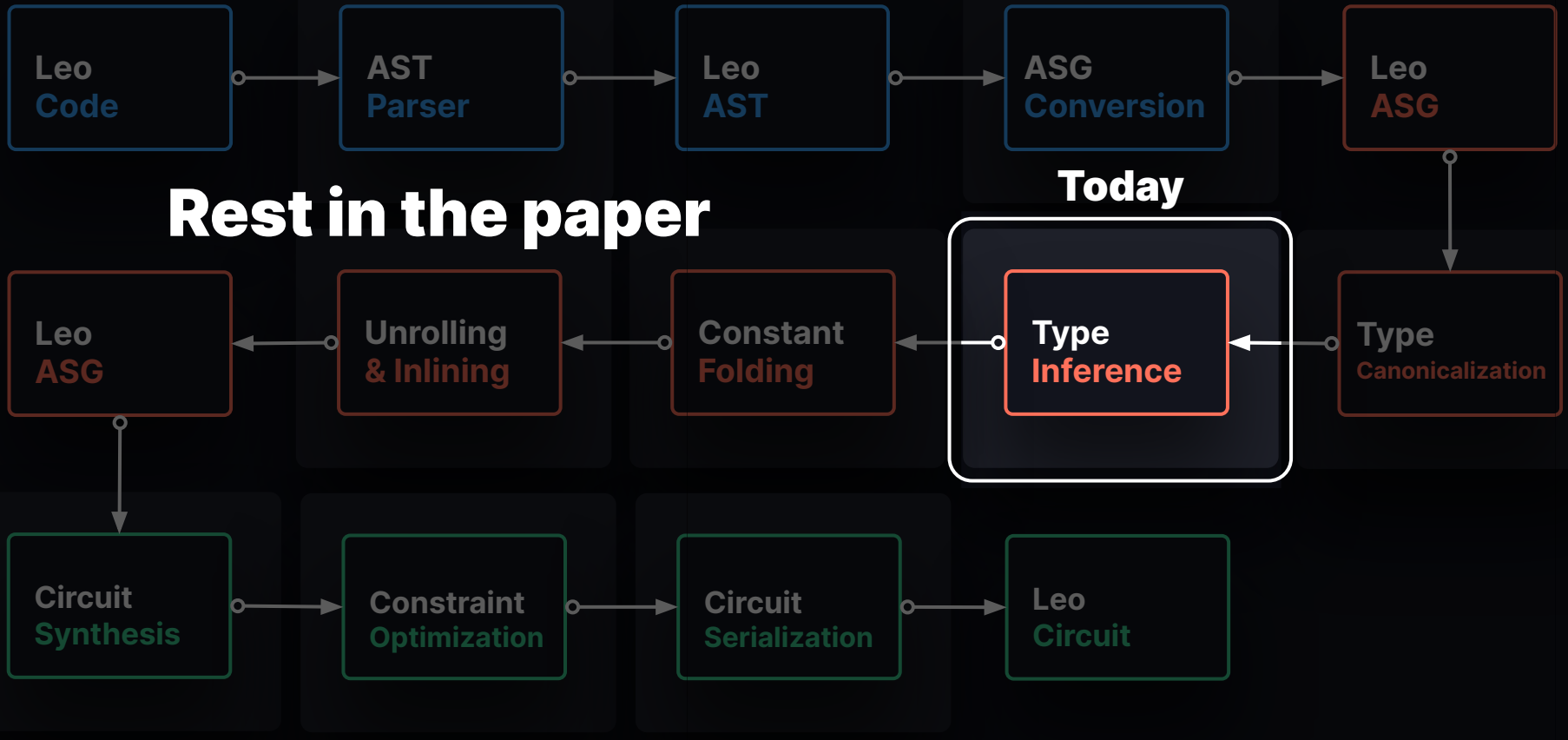
Verifiably Computed
Executions can be succinctly verified by anyone

Correct-by-Construction
Compiled programs are formally verified

The Leo Compiler



The Leo Compiler



Type Inference

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

```
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 {
    ...
}
```

Type Inference

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

```
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

Step One

```
function foo() {
    let a: T1 = 3u8;
    let b: i16 = -5T2;
    let c: T3 = 100T4;
    let d: T5 = 100T6;
    let e: T7 = [c, 8i8];
    let f: T8 = 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: T1 = 3u8;
    let b: i16 = -5T2;
    let c: T3 = 100T4;
    let d: T5 = 100T6;
    let e: T7 = [c, 8i8];
    let f: T8 = bar(d);
}

function bar(g: field) -> bool {
    ...
}
```

Step Two

```
T1  ≡ u8
i16 ≡ T2
T3  ≡ T4
T5  ≡ T6
T7  ≡ [T3; 2]
T5  ≡ field
T8  ≡ bool
```

Form Equations

Step Three

```
T1 = u8
T2 = i16
T3 = i8
T4 = i8
T5 = field
T6 = field
T7 = [i8; 2]
T8 = bool
```

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?

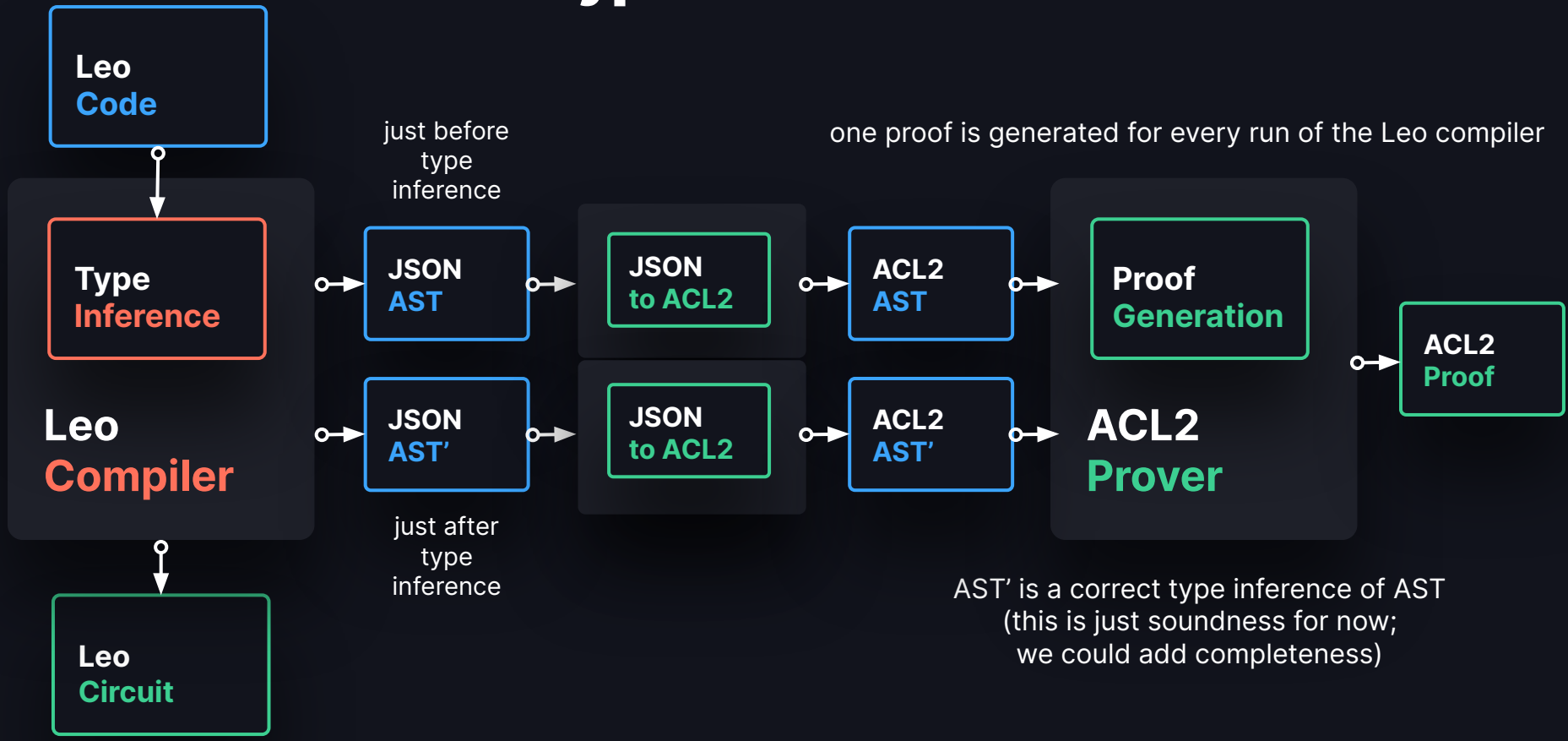
Type
Inference

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

Formal Methods

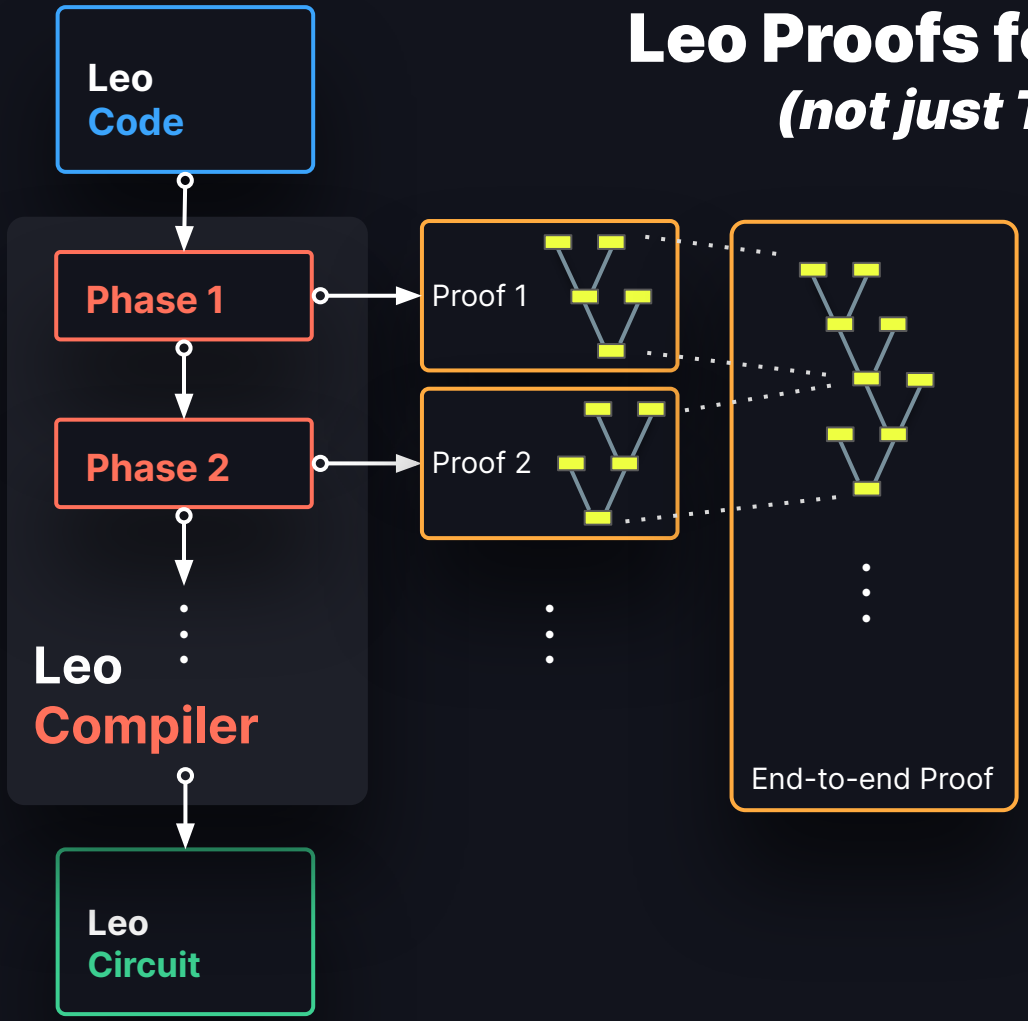
- Formal methods are techniques and tools based on formal logic that aim to establish desired properties of programs by way of mathematical proofs.
- We use the **industrial-strength ACL2 theorem prover** in Leo.
- Leo is a **verifying compiler** - one that generates, every time it is run, a proof of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

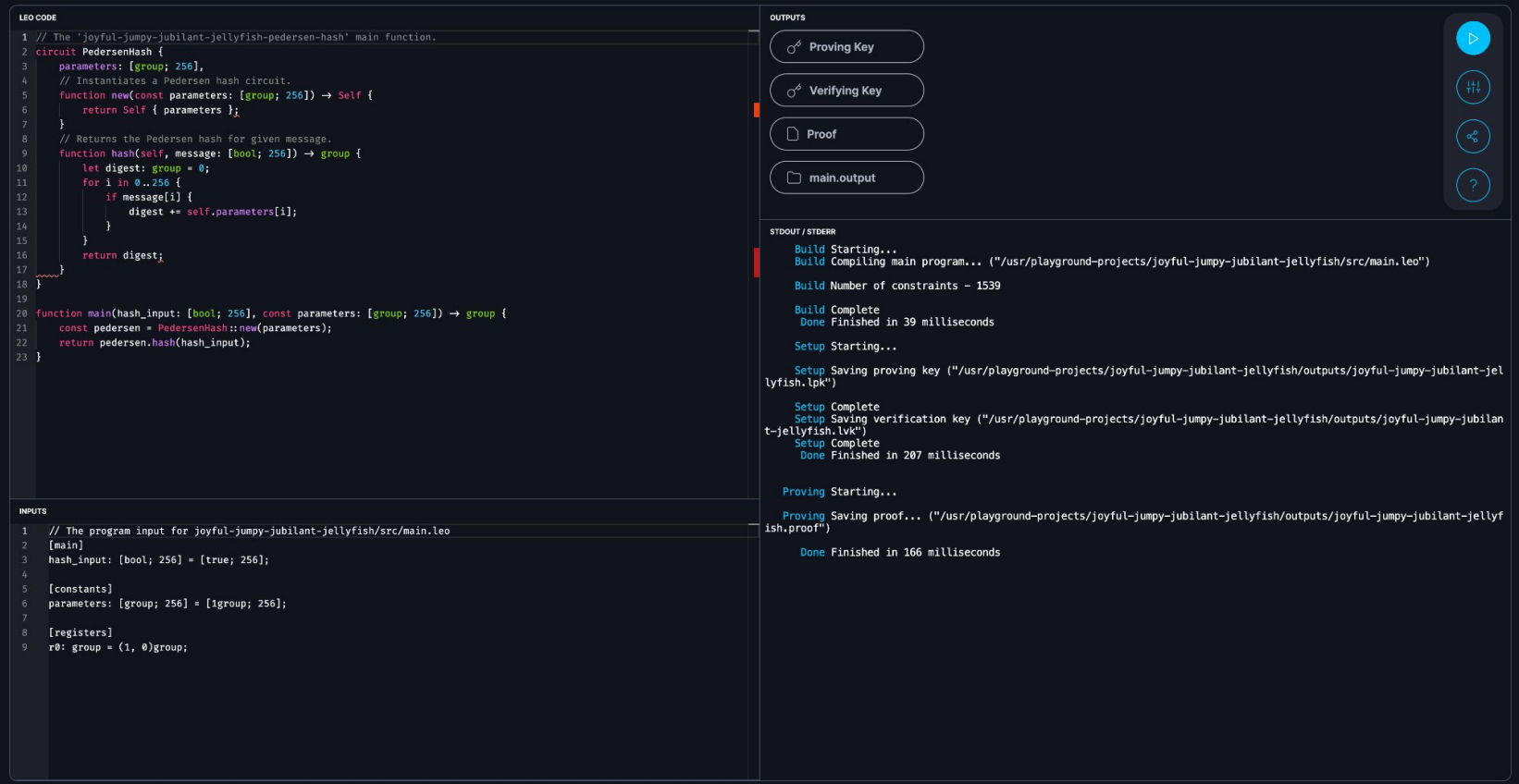
Leo programs are formally verified in ACL2.
Open-source (soon), partial passes complete

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



play.leo-lang.org

Aleo Studio

Subscribe

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

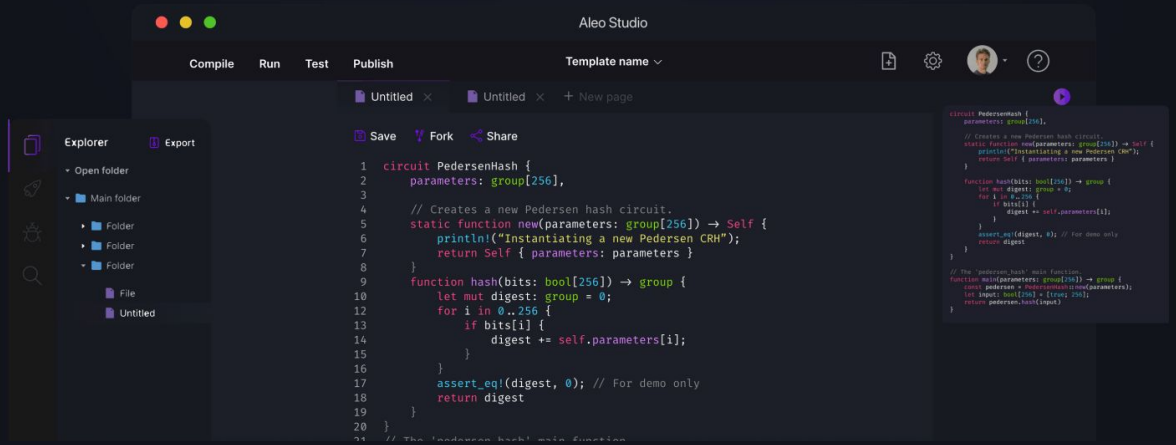
Welcome to Aleo Studio.

Imagine a world of truly private applications.

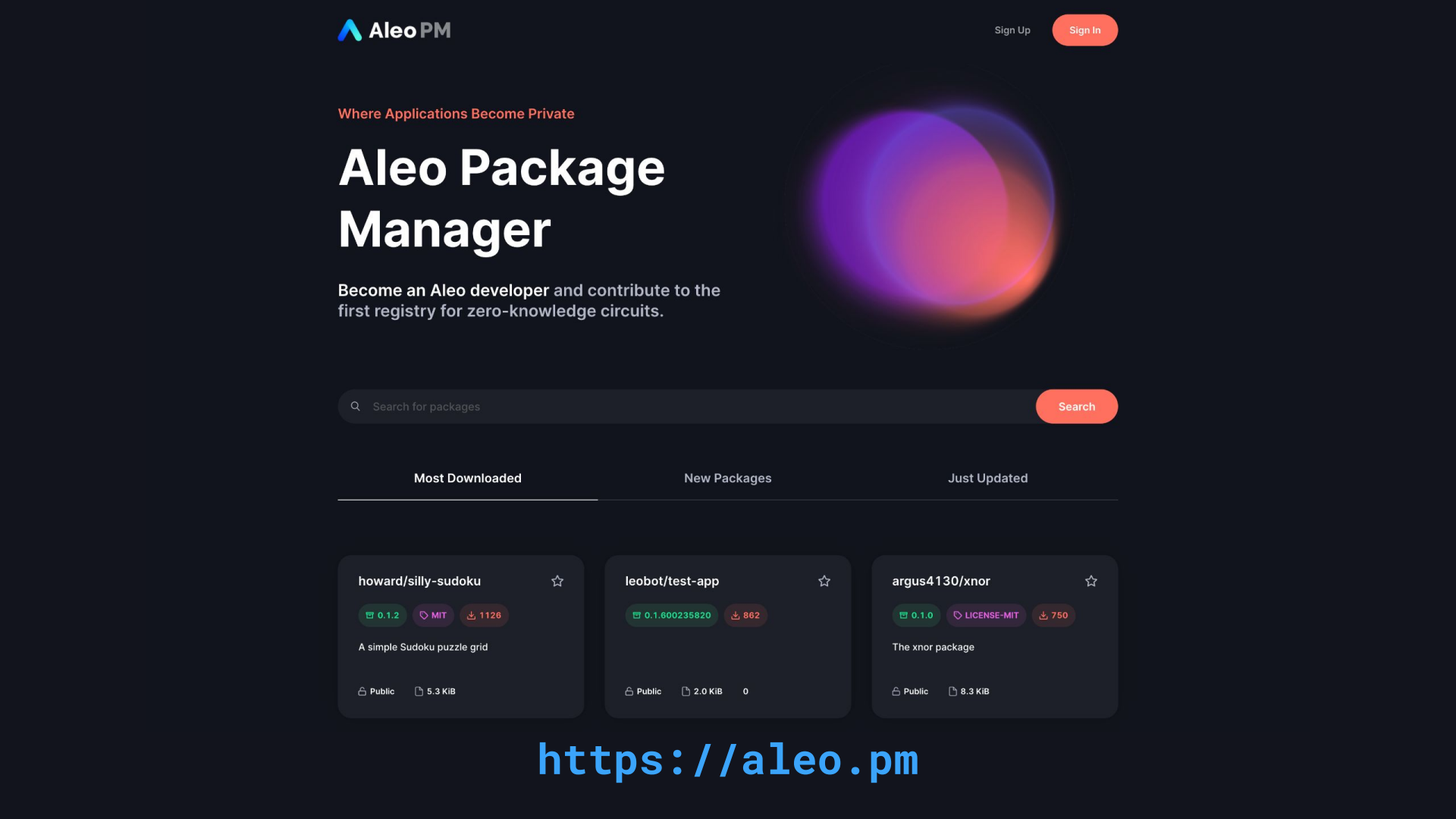
Windows Linux macOS

Download Alpha

macOS 10.13.0 and newer



https://aleo.studio

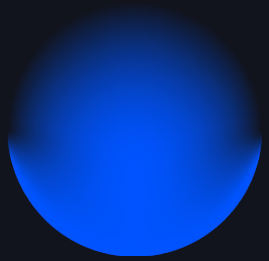


https://aleo.pm

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