Linea

Nov 13th

Devcon 7

Wizard

Build your own P-IOP protocol in 15 min!

Linea

Nov 13th

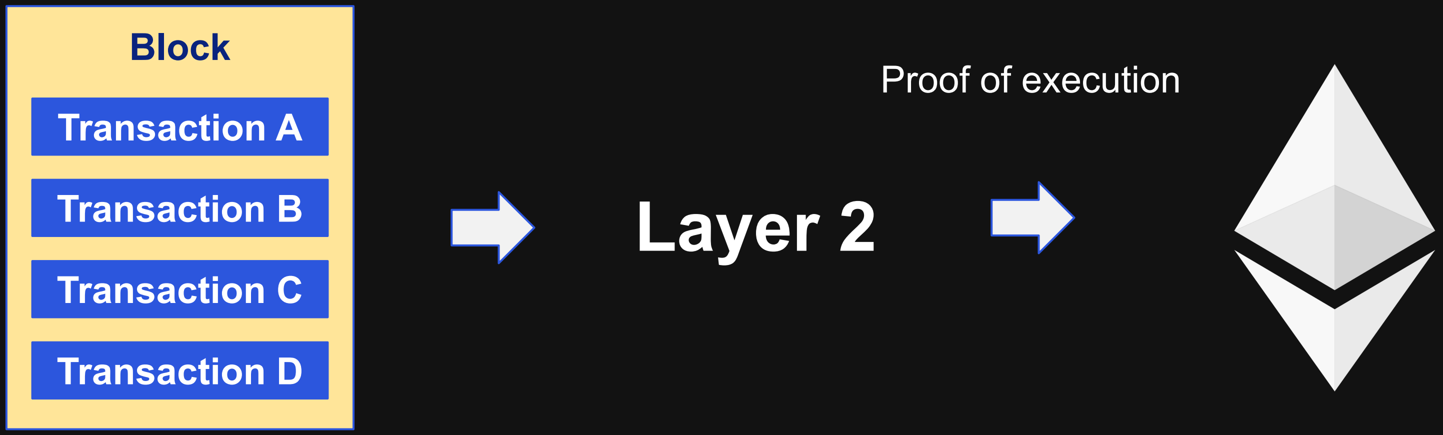
Devcon 7

Hello everyone

Alexandre Belling

Linea | Prover team Lead

SNARKS for scalability



ZKP and SNARK

ZK - Argument/Proof of knowledge

- **Computational**/**Statistical** Knowledge Soundness
- Completeness
- **Zero-Knowledge**

ZK - SNARKs

- **ZK - Argument of Knowledge**
- **Succinctness**
- **Non-interactivity**

Proving execution is complicated

- Formally, proving execution is equivalent to proving knowledge of an execution trace satisfying the EVM specification.

- The involves a very large number of constraints of various kinds. And there are a lot of edge-cases to consider.

- We need a proof system that is at the same time extremely flexible and also extremely efficient.

- And that’s how we created the Wizard framework

The Wizard framework

- Developed to fit Linea's requirements to build a zk-EVM
- The framework allows to specify protocols in a modular fashion

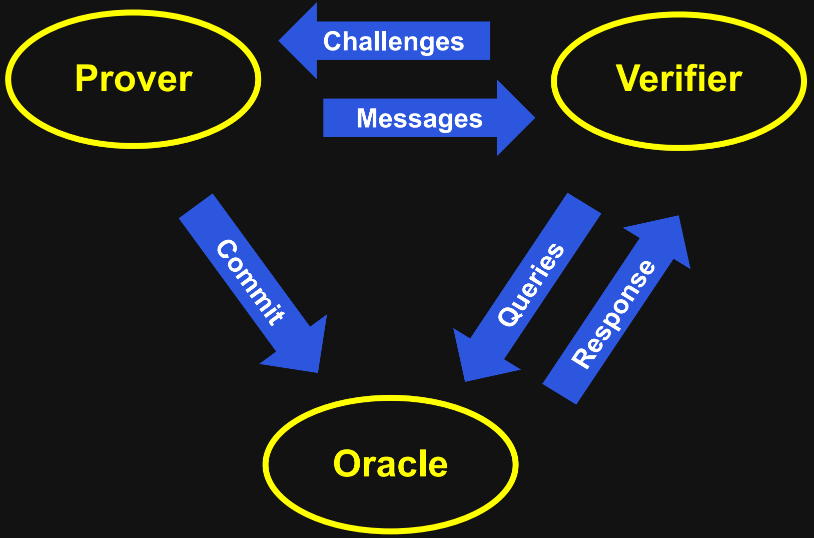
Powerful

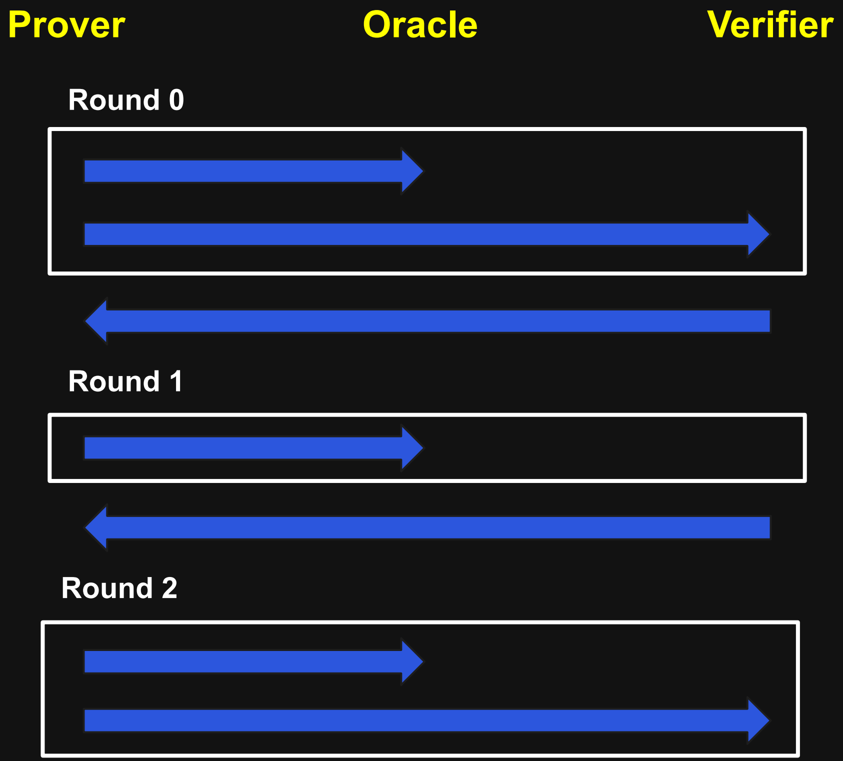
- A wide range of possible constraints and queries are available. They generalize most security paradigm (IOP, P-IOP, …) and constraints systems
- The framework secure automates random coins sampling

Modular

- Users can extend it by adding their own type of constraints sub-arguments and by specifying a compiler routine

The model


Prover
Challenges
Messages
Verifier
Commit
Queries
Response
Oracle


Prover
Oracle
Verifier
Round 0
Round 1
Round 2

First, we have columns

A column is a vector of field element taking part in the protocol. It is defined

- **Visibility:** A column can be either internal to the prover, public to the verifier or sent to the oracle. It can also be precomputed or on-line. Most columns have the “committed” visibility, indicating they are to be submitted to the oracle.
- **Size:** Columns have a prespecified size
- **Name:** Columns
- **Round:** An interaction round at which the column is instantiated in the protocol

```
IsActive:       comp.InsertCommit(0, "MODEXP_IS_ACTIVE", size),
Limbs:          comp.InsertCommit(0, "MODEXP_LIMBS", size),
IsSmall:        comp.InsertCommit(0, "MODEXP_IS_SMALL", size),
IsLarge:        comp.InsertCommit(0, "MODEXP_IS_LARGE", size),
LsbIndicator:   comp.InsertPrecomputed("MODEXP_LSB_INDICATOR", lsbIndicatorValue(size)),
ToSmallCirc:    comp.InsertCommit(0, "MODEXP_TO_SMALL_CIRC", size),
```

Queries

Queries are a "question" the the verifier may ask to the oracle about committed columns. For example:

- "If I reinterpret column A as a polynomial in Lagrange basis, how does it evaluate to point x?"
- "What is the inner-product between these two committed columns?"
- "What the is the value of the 10th position of this column?"

```
comp.InsertGlobal(
    0,
    "STATE_SUMMARY_ADDRESS_CAN_ONLY_INCREASE_IN_BLOCK_RANGE",
    sym.Mul(
        ss.IsActive,
        sym.Sub(
            ss.BatchNumber,
            column.Shift(ss.BatchNumber, -1),
            1,
        ),
        sym.Add(
            ss.Account.HasGreaterAddressAsPrev,
            ss.Account.HasSameAddressAsPrev,
            -1,
        ),
    ),
)
```

And they can be Yes/No questions too:

- Is this arithmetic constraints/ lookup constraints satisfied?

Linea

Nov 13th

Devcon 7

Compilers

Wizard protocol

Compilers

Protocol in the standard model

```go
return func(comp *wizard.CompiledIOP) {
	specialqueries.RangeProof(comp)
	specialqueries.CompileFixedPermutations(comp)
	permutation.CompileGrandProduct(comp)
	lookup.CompileLogDerivative(comp)
	innerproduct.Compile(comp)
	if withLog_ {
		logdata.Log("after-expansion")(comp)
	}
	sticker.Sticker(minStickSize, targetColSize)(comp)
	splitter.SplitColumns(targetColSize)(comp)
	if withLog_ {
		logdata.Log("post-rectangularization")(comp)
	}
	cleanup.CleanUp(comp)
	localcs.Compile(comp)
	globalcs.Compile(comp)
	univariates.CompileLocalOpening(comp)
	univariates.Naturalize(comp)
	univariates.MultiPointToSinglePoint(targetColSize)(comp)
	if withLog_ {
		logdata.Log("end-of-arcane")(comp)
	}
}
```

Let’s implement a proof system for Plonk CS

Witness structure and gate constraints:

general.

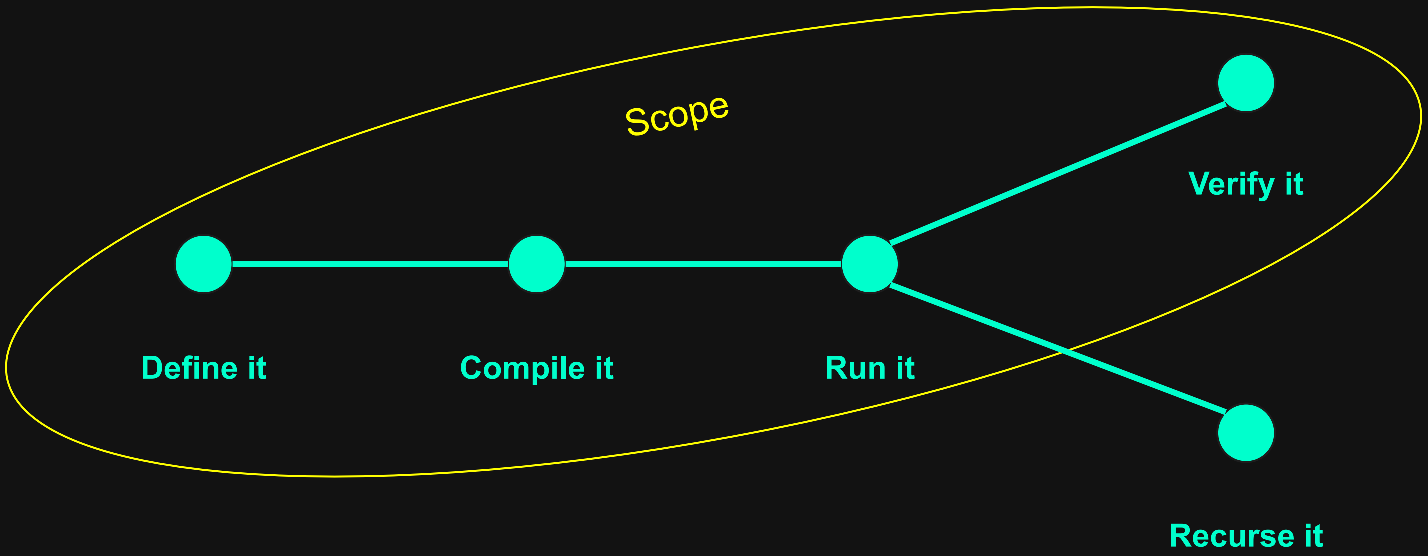
The constraint system $\mathscr{C} = (\mathcal{V}, \mathcal{Q})$ is defined as follows.

- $\mathcal{V}$ is of the form $\mathcal{V} = (\mathbf{a}, \mathbf{b}, \mathbf{c})$, where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in [m]^n$. We think of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ as the left, right and output sequence of $\mathscr{C}$ respectively.
- $\mathcal{Q} = (\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}) \in (\mathbb{F}^n)^5$ where we think of $\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C} \in \mathbb{F}^n$ as “selector vectors”.

We say $\mathbf{x} \in \mathbb{F}^m$ *satisfies* $\mathscr{C}$ if for each $i \in [n]$,

$$(\mathbf{q_L})_i \cdot \mathbf{x}_{\mathbf{a}_i} + (\mathbf{q_R})_i \cdot \mathbf{x}_{\mathbf{b}_i} + (\mathbf{q_O})_i \cdot \mathbf{x}_{\mathbf{c}_i} + (\mathbf{q_M})_i \cdot (\mathbf{x}_{\mathbf{a}_i} \mathbf{x}_{\mathbf{b}_i}) + (\mathbf{q_C})_i = 0.$$

Workflow



Defining your protocol

The first step is to construct a blueprint of the protocol. This is done via a “Define” function.

The function specifies

- The columns existing in the protocol.
- Which coins exists in the protocol
- All the different queries
- Some specific verifier checks

```go
func Define(bui *wizard.Builder) {
    // [...]
}
```

Let’s create the columns firsts

```go
plk := &TinyPlonkCS{
    Ql:       comp.InsertPrecomputed("QL", ql),
    Qr:       comp.InsertPrecomputed("QR", qr),
    Qo:       comp.InsertPrecomputed("QO", qo),
    Qm:       comp.InsertPrecomputed("QM", qm),
    Qc:       comp.InsertPrecomputed("QC", qc),
    Xa:       comp.InsertCommit(0, "XA", nbConstraints),
    Xb:       comp.InsertCommit(0, "XB", nbConstraints),
    Xc:       comp.InsertCommit(0, "XC", nbConstraints),
    Pi:       comp.InsertProof(0, "PI", nbConstraints),
    NbPublic: nbPublic,
}
```

Round number

*Note: this is for several instance of the same circuit**

Linea

Nov 13th

Devcon 7

Declare the queries

```go
// This defines the gate constraint
comp.InsertGlobal(
    0,
    "GATE-CS",
    sym.Add(
        sym.Mul(plk.Ql, plk.Xa),
        sym.Mul(plk.Qr, plk.Xb),
        sym.Mul(plk.Qm, plk.Xa, plk.Xb),
        sym.Mul(plk.Qo, plk.Xc),
        plk.Qc,
        plk.Pi,
    ),
)
```

```go
// This declares the copy constraints
comp.InsertFixedPermutation(
    0,
    "COPY-CS",
    []sv.SmartVector{sa, sb, sc},
    []ifaces.Column{plk.Xa, plk.Xb, plk.Xc},
    []ifaces.Column{plk.Xa, plk.Xb, plk.Xc},
)
```

Adds a missing a verifier check

```go

// This implements the [wizard.VerifierAction] interface
func (v *verifierDirectCheck) Run(run *wizard.VerifierRuntime) error {

    pi := v.Pi.GetColAssignment(run)

    for i := v.NbPublic; i < pi.Len(); i++ {
        x := pi.Get(i)
        if !x.IsZero() {
            return fmt.Errorf("PI is not well-formed")
        }
    }

    return nil
}

```

```go
// This tells the verifier to check the well-formedness of the PI vector
comp.RegisterVerifierAction(0, &verifierDirectCheck{TinyPlonkCS: *plk})

```

Compiling into an actual protocol

```go
comp := wizard.Compile(
    // our "Define" define function
    define,
    // A list of compiler steps to construct the proof system
    specialqueries.RangeProof,
    specialqueries.CompileFixedPermutations,
    permutation.CompileGrandProduct,
    lookup.CompileLogDerivative,
    innerproduct.Compile,
    cleanup.CleanUp,
    localcs.Compile,
    globalcs.Compile,
    univariates.CompileLocalOpening,
    univariates.Naturalize,
    univariates.MultiPointToSinglePoint(64),
    vortex.Compile(2),
)
```

Remains to write the prover!

This is done by specifying a “Prove” function.

```go
// Prove is responsible for assigning the columns we created and assigning and
// assigning query parameters (for instance, the value of X if there is a
// univariate query)
func Prove(run *wizard.ProverRuntime) {

    // [...]
}
```

And that's all we need to do

```go
// Assign allows assigning the columns of the receiver [TinyPlonk] so that we
// construct the witness of a proof to generate. The caller is responsible to
// provide a solution to the Plonk circuit.
func (plk *TinyPlonkCS) Assign(run *wizard.ProverRuntime, xa, xb, xc, pi sv.SmartVector) {
	run.AssignColumn(plk.Xa.GetColID(), xa)
	run.AssignColumn(plk.Xb.GetColID(), xb)
	run.AssignColumn(plk.Xc.GetColID(), xc)
	run.AssignColumn(plk.Pi.GetColID(), pi)
}
```

Wrap our code with gnark's frontend

```go
// DefineFromGnark constructs a [TinyPlonk] taking as inputs a [frontend.Circuit].
// It is more friendly to use than writting the constraints by hands and calling
// [DefineRaw].
func DefineFromGnark(comp *wizard.CompiledIOP, circ frontend.Circuit) *TinyPlonkCS {

    ccs, err := frontend.Compile(ecc.BLS12_377.ScalarField(), scs.NewBuilder, circ)
    if err != nil {
        utils.Panic("error
    }
```

```go
// AssignFromGnark constructs and assign a witness from a gnark circuits assignment.
// It is a wrapper around [Assign] that is simpler to use [Assign] as it performs
// the resolution of the assignment via the gnark solver and remove that burden
// from the caller.
//
// The receiver **must** have been constructed via the [DefineFromGnark] for this
// to work.
func (plk *TinyPlonkCS) AssignFromGnark(run *wizard.ProverRuntime, a frontend.Circuit) {

    if plk.spr == nil {
        utils.Panic("the [%T] must have been constructed via [DefineWithGnark]", plk)
    }
```

Let’s write a circuit

```go
// FibonacciCircuit is a circuit enforcing that U50 is the 50-th term of a
// sequence defined by the recursion U[i+2] = U[i+1] + U[i] given U[0] and U[1].
You, 40 minutes ago | 1 author (You)
type FibonacciCircuit struct {
    // U0, U1 are the initial values of a Fibonacci sequence
    U0, U1 frontend.Variable `gnark:",public"`
    // U50 is the 50-th term of the sequence
    U50 frontend.Variable `gnark:",public"`
}
```

```go
// Define implements the [frontend.Circuit] interface
func (f *FibonacciCircuit) Define(api frontend.API) error {

    var (
        prevprev = f.U0
        prev     = f.U1
    )

    for i := 2; i <= 50; i++ {
        prevprev, prev = prev, api.Add(prevprev, prev)
    }

    api.AssertIsEqual(prev, f.U50)

    return nil
}
```

And let's finally run it!

```go

	prove := func(run *wizard.ProverRuntime) {
		plk.AssignFromGnark(run, &FibonacciCircuit{
			U0:  0,
			U1:  1,
			U50: fibo(field.Zero(), field.One(), 50),
		})
	}

	var (
		proof = wizard.Prove(comp, prove)
		err   = wizard.Verify(comp, proof)
	)

	if err != nil {
		t.Fatalf("the verification failed: %v", err)
	}
```

Future improvements

- A new simplified API
- More types of queries
- Support for small fields
- Sumcheck-based compilation suite
- Support for columns of unlimited sizes
- A more comprehensive standard package

Checkout the code

https://github.com/Consensys/linea-monorepo/tree/devcon/tiny-plonk/prover/example/tinyplonk