Grant Report: Ethereum Foundation

Extraction of Zero-Knowledge circuits in LLZK to Rocq

Date: 2025-07-31

To: Ethereum Foundation https://ethereum.foundation/

From: Formal Land (Arae) https://formal.land/

Contact: guillaume.claret@formal.land

> This grant report is to answer the task: "Demonstrate a Plonky3 → Rocq extraction via `coq-of-rust`" that we adapt to target LLZK instead, with a mechanized translation to Rocq and the verification of some examples.

Summary

We have built a translation tool from a subset of the LLZK language to the Rocq formal verification language. This tool is built in C++ as an additional pass in the LLZK infrastructure, which does nothing but print the Rocq translation. The code is available in this pull request: https://github.com/formal-land/llzk-lib/pull/1

We have focused on the translation of this example file test/Analysis/constraint_dependency_graph_pass.llzk, which is about 500 lines long excluding comments. It uses various LLZK features, including the manipulation of structures, arrays, function calls, and polymorphism.

We successfully translated this file in Rocq to the code in Garden/LLZK/translated.v, which is of a similar size compared to the original `.llzk` file. We define a semantics for the LLZK operators in Garden/LLZK/M.v. We specify and formally verify all the definitions in this example to validate our reasoning rules, in the file Garden/LLZK/verification.v.

We wrote three blog posts to explain our work and welcome people to discuss:

- 🥷 Beginning of a formal verification tool for LLZK
- 🥷 Semantics for LLZK in Rocq
- 🥷 Formal verification of LLZK circuits in Rocq

These blog posts provide more technical information about our translation and verification process.

As future work, we plan to add more automation on the Rocq side to handle the repetitive steps in the proofs and increase the support for the LLZK language. An example of a construct that we do not handle yet is building arrays of a dynamic size, for example, in a `for` loop. A solution is to unroll the loops, but this is not always the most convenient for the verification process.

Translation

The translation is done in a single pass in C++, pretty-printing the Rocq translation on the fly. We indent the output to make it readable. Most of the lines are short, as in SSA form, only one operation is applied per line.

We had trouble printing the names of the variables, as this information is not available in the AST. For that, we reuse pretty-printing facilities from the MLIR library, which does the work to generate reasonable and non-conflicting names. This part is still a little bit fragile and slow on large files (thousands of lines).

Here is an LLZK example that we will translate in this report:

```
function.def @global_add(%a: !felt.type, %b: !felt.type) -> !felt.type {
  %c = felt.add %a, %b
  function.return %c : !felt.type
}

struct.def @Adder {
  struct.field @sum : !felt.type {llzk.pub}

  function.def @compute(%a: !felt.type, %b: !felt.type) -> !struct.type<@Adder>
    %self = struct.new : !struct.type<@Adder>
    %sum = function.call @global_add(%a, %b) :
      (!felt.type, !felt.type) -> (!felt.type)
    struct.writef %self[@sum] = %sum : !struct.type<@Adder>, !felt.type
    function.return %self : !struct.type<@Adder>
  }

  function.def @constrain(
      %self: !struct.type<@Adder>, %a: !felt.type, %b: !felt.type
  ) {
    %sum = struct.readf %self[@sum] : !struct.type<@Adder>, !felt.type
    %c = function.call @global_add(%a, %b) :
```

```
      (!felt.type, !felt.type) -> (!felt.type)
    constrain.eq %sum, %c : !felt.type
    function.return
  }
}
```

Representation

We make a shallow embedding of LLZK in Rocq, using purely functional definitions for as many features as possible, including arrays, structures, and field elements manipulations, as well as `for` loops which must be bounded. We still have some side effects, to enforce a polynomial constraint, or mutate arrays or structures. We encode these side effects in a free monad.

Here is the generated Rocq translation of the example above:

```
Definition global_add {p} `{Prime p}
    (arg_fun_0 : Felt.t) (arg_fun_1 : Felt.t) :
    M.t Felt.t :=
  let var_0 : Felt.t := BinOp.add arg_fun_0 arg_fun_1 in
  M.Pure var_0.

Module Adder.
  Record t : Set := {
    sum : Felt.t;
  }.

  Global Instance IsMapMop {ρ} `{Prime ρ} : MapMod t := {
    map_mod α := {|
      sum := map_mod α.(sum);
    |};
  }.

  Definition constrain {p} `{Prime p}
      (arg_fun_0 : Adder.t) (arg_fun_1 : Felt.t) (arg_fun_2 : Felt.t) :
      M.t unit :=
    let var_0 : Felt.t := arg_fun_0.(Adder.sum) in
    let* var_1 : Felt.t := global_add arg_fun_1 arg_fun_2 in
    let* _ : unit := M.AssertEqual var_0 var_1 in
    M.Pure tt.

  Definition compute {p} `{Prime p}
      (arg_fun_0 : Felt.t) (arg_fun_1 : Felt.t) :
      M.t Adder.t :=
    let* var_self : Adder.t := M.CreateStruct in
    let* var_0 : Felt.t := global_add arg_fun_0 arg_fun_1 in
    let* _ : unit := M.FieldWrite var_self.(Adder.sum) var_0 in
    M.Pure var_self.
End Adder.
```

Here are the monad primitives that we currently have in Rocq:

```
Module M.
  Inductive t : Set -> Set :=
  | Pure {A : Set} (value : A) : t A
  | AssertEqual {A : Set} (x1 x2 : A) : t unit
  | AssertIn {A : Set} {Ns : list nat} (x : A) (array : Array.t A Ns) : t unit
  | CreateStruct {A : Set} : t A
  | FieldWrite {A : Set} (field : A) (value : A) : t unit
  | Let {A B : Set} (e : t A) (k : A -> t B) : t B.
End M.
```

Here is a quick explanation of the primitives:

- `Pure` is the return of a value
- `AssertEqual` is the equality constraint (of two field elements)
- `AssertIn` is the membership constraint (of a field element in an array)
- `CreateStruct` is the creation of a structure (undefined for now)
- `FieldWrite` is forcing the field of a structure to be equal to a given value
- `Let` is the monadic bind

Reasoning rules

We use the usual reasoning principles for the purely functional part of the translation. For the side effects, we have the following rules in Rocq:

```
Module Run.
  Reserved Notation "{{ e 🔽 output , P }}".

  Inductive t : forall {A : Set}, M.t A -> A -> Prop -> Prop :=
  | Pure {A : Set} (value : A) :
    {{ M.Pure value 🔽 value, True }}
  | AssertEqual {A : Set} (x1 x2 : A) :
    {{ M.AssertEqual x1 x2 🔽 tt, x1 = x2 }}
  | AssertIn {A : Set} {Ns : list nat} (x : A) (array : Array.t A Ns) :
    {{ M.AssertIn x array 🔽
      tt,
      exists indexes, Array.MultiIndex.Valid.t indexes /\
      Array.read array indexes = x
    }}
  | CreateStruct {A : Set} (value : A) :
    {{ M.CreateStruct 🔽 value, True }}
  | FieldWrite {A : Set} (field : A) :
    {{ M.FieldWrite field field 🔽 tt, True }}
  | Let {A B : Set}
      (e : M.t A) (k : A -> M.t B) (value : A) (output : B) (P1 P2 : Prop) :
    {{ e 🔽 value, P1 }} ->
    (P1 -> {{ k value 🔽 output, P2 }}) ->
    {{ M.Let e k 🔽 output, P1 /\ P2 }}
  | Implies {A : Set} (e : M.t A) (value : A) (P1 P2 : Prop) :
    {{ e 🔽 value, P1 }} ->
    (P1 -> P2) ->
```

```
    {{ e 🔽 value, P2 }}

  where "{{ e 🔽 output , P }}" := (t e output P).
End Run.
```

The notation:

```
{{ e 🔽 output , P }}
```

means that we can reduce the monadic expression `e` to the value `output` and that the property `P` is true if we validate all the constraints (meaning that the circuit validates the witness).

Our goal will be to show that we can evaluate the monadic expression of a circuit with a property `P` of the form:

```
outputs = expected_outputs_of inputs
```

to ensure that the circuit has no under-constraints.

Here is a short explanation of each rule:

- `Pure`: we directly return the purely functional value and add no constraint
- `AssertEqual`: we add the equality constraint between the two arguments, and return the unit value `tt`
- `AssertIn`: we add the membership constraint between an element and an array
- `CreateStruct`: we create a structure that is declared in the code but not defined yet. Here, the value of the structure is a "prophecy" and must be correctly guessed in the proof in order not to get stuck later in the proof.
- `FieldWrite`: we force, or check depending on the interpretation, the field of a structure to be equal to a given value
- `Let`: we bind two monadic expressions, returning the conjunction of the two properties for the constraints
- `Implies`: we can rewrite the property `P` to a nicer formula, as long as this is a property that we imply with the current one. Note that we are interested in verifying an implication but not an equivalence, as this is enough to ensure that the circuit has no under-constraints.

Example proof

We specify and verify the example above to ensure that the `constrain` function cannot validate a witness that is not the one generated by the `compute` function (no under-

constraints). We have done this exercise of specification and verification for the whole example file that we are translating, which is about 500 lines long.

The formal specifications and proofs of the examples above are as follows. More details are available in the blog posts.

Auxiliary function `global_add`

```
Lemma global_add_eq {p} `{Prime p} (x y : Felt.t) :
  {{ global_add x y ▼ ((x + y) mod p)%Z, True }}.
Proof.
  apply Run.Pure.
Qed.
```

Compute function

```
Lemma compute_eq {p} `{Prime p} (x y : Felt.t) :
  {{ Adder.compute x y ▼
    spec x y, True
  }}.
Proof.
  unfold Adder.compute.
  eapply Run.Implies. {
    eapply Run.Let. {
      eapply Run.CreateStruct with (value := Adder.Build_t _).
    }
    intros _.
    eapply Run.Let. {
      apply global_add_eq.
    }
    intros _.
    eapply Run.Let. {
      eapply Run.FieldWrite.
    }
    intros _.
    apply Run.Pure.
  }
  easy.
Qed.
```

Constrain function

```
Lemma contrain_implies {p} `{Prime p}
    (self : Adder.t)
    (x y : Felt.t) :
  let self := map_mod self in
  {{ Adder.constrain self x y ▼
    tt,
    self = spec x y
  }}.
Proof.
  unfold Adder.constrain.
```

```
  eapply Run.Implies. {
    eapply Run.Let. {
      apply global_add_eq.
    }
    intros _.
    eapply Run.Let. {
      apply Run.AssertEqual.
    }
    intros _.
    apply Run.Pure.
  }
  unfold spec.
  hauto lq: on.
Qed.
```

Conclusion

We are thankful to the Ethereum Foundation for funding this work and this opportunity, and we are open to any feedback or discussion.

In the future, we plan to continue extending the support for the LLZK language and to add more automation on the proof side in Rocq.