Formal Verification of Keccak in Plonky3

Grant Report for the Ethereum Foundation

2025-11-13

Formal Land

Contents

1 Introduction

We formally verified the circuit implementation of the Keccak algorithm in Plonky3 for the following two properties:

- Determinism (absence of under-constraints);
- Correctness, that is to say, we compute the same result as a simpler definition of Keccak using direct boolean computations.

We have not covered completeness, which is the existence of at least one solution for the circuit, or the correctness of the generator function in Rust in Plonky3 to build at least one solution to the system.

We verified the Keccak circuits at the implementation level, using the Garden formal verification framework in Rocq that we developed for this grant. We have not found any issues in the circuit, but we believe the `preimage` column could be removed to save some constraints.

Here is the link to our verification work: `https://github.com/formal-land/garden/tree/main/Garden/Plonky3/keccak`

The important files are:

- columns.v with the definition of the data types on which we apply the circuit constraints
- air.v the constraints of the Keccak circuit, as defined in Rocq, and following the structure of the Plonky3 code
- pretty_print.v the code to display the low-level constraints from the Rocq circuit
- air.snapshot the low-level constraints; this file contains around 100.000 lines
- proofs/air.v the formal specification and verification of the Keccak circuit

We made a fork of Plonky3 to pretty-print the low-level constraints in:

`https://github.com/formal-land/Plonky3/pull/1`

The code we are verifying is the Keccak circuits defined in the folder:

`https://github.com/Plonky3/Plonky3/tree/main/keccak-air/src`

All of our work is under the open-source MIT license.

2 Model of the circuits

We model the circuits with a shallow embedding using our Garden monad, which provides primitives such as the equality assertion between two field expressions. We use the plain $\mathbb{Z}$ type of Rocq to express field elements. For the values of the variables of the circuits, we first quantify universally over $\mathbb{Z}$ and then take the value modulo a prime number p.

Here are the primitives of the monad that we use for the circuits:

```
Module M.
  Inductive t (A : Set) : Set :=
  | Pure (value : A) : t A
  | AssertZero (x : Z) (value : A) : t A
  | Let {B : Set} (e : t B) (k : B -> t A) : t A
  | When (condition : Z) (e : t A) : t A
  | Message (message : string) (k : t A) : t A.
End M.
```

The monadic operations are `Pure` and `Let`, and `AssertZero` and `When` are similar to the Plonky3 operators. We use `Message` to log information when displaying the low-level constraints, to help in debugging that they are the same as in Rust.

Our predicate to describe the behavior of circuits is:

$$\{\{\ e\ \triangledown\ \text{output},\ P\ \}\}$$

with:

- e the expression representing the circuit;
- output the expression representing the result of the circuit, very often the unit value when the circuit is of type `void` in Rust, and sometimes field expressions;
- P the predicate implied on the current free variables by the execution of the circuit.

Here is an example of a circuit model in Rocq:

```
M.for_in_zero_to_n 5 (fun y =>
M.for_in_zero_to_n 5 (fun x =>
  M.when first_step (
    M.assert_zeros (N := U64_LIMBS) {|
      Array.get limb :=
        local.(KeccakCols.preimage).[y].[x].[limb] -F
        local.(KeccakCols.a).[y].[x].[limb]
    |}
  )
```

corresponding to the Rust code in Plonky3:

```
for y in 0..5 {
    for x in 0..5 {
        builder
            .when(first_step)
            .assert_zeros::<U64_LIMBS, _>(array::from_fn(|limb| {
                local.preimage[y][x][limb] - local.a[y][x][limb]
            }));
    }
}
```

As you can see, the two programs follow the same structure. To make sure that they define the same constraints, we unroll all the loops and pretty-print the polynomial constraints on both sides, Rocq and Rust. On the Rust side, we use the fact that Plonky3 is highly parametrizable to provide a custom AIR builder printing the constraints instead of proving the circuit.

On the Rocq side, this is not directly possible as we use a shallow embedding: for example, polynomials are in $\mathbb{Z}$ instead of being a tree structure of binary operators, like in Rust. We made this choice to simplify the verification work in Rocq, using the $\mathbb{Z}$ type to represent both the constants, the variables, and the expressions. So we first run a tactic (Rocq meta-programming) to extract the deep-embedding from the shallow definitions, and then use a pretty-printing similar to the one in Rust. Comparing the constraints of our Rocq definition with the Rust implementation, we discovred a few mistakes that we fiexed. A video on *X* gives more details for the pretty-printing of the Rocq model: https://x.com/FormalLand/status/1978567174550667448

3 Specification of the code

We run a first analysis of the code, going step-by-step through each loop of Keccak to extract a logical formula summarizing the constraints. Most of the operations are about manipulating arrays of booleans or arrays of limbs, which are a more compact representation of arrays of booleans. For example, for the loop above, we verify that it implies the constraints:

```
Module Spec.
  Definition t (local : KeccakCols.t) : Prop :=
    local.(KeccakCols.step_flags).[0] <> 0 ->
    local.(KeccakCols.preimage) =F local.(KeccakCols.a).
End Spec.
```

For some of the loops that are using clever tricks to represent boolean operations, we run an additional verification step to prove a simpler expression of the constraints once we combine all the loops. In particular, we show that we can express the constraints as deterministic expressions of previously constrained variables.

An interesting property is the following, where we show that we are computing XOR boolean operations using equations with differences and products. We prove this property by expanding the formula for all possible boolean values that appear in it.

```
Lemma xor_sum_diff_eq {p} `{Prime p} (H_p : 6 <= p)
    (local : KeccakCols.t) (x z : Z)
    (H_a_prime_bools : IsBool.t local.(KeccakCols.a_prime))
    (H_c_prime_bools : IsBool.t local.(KeccakCols.c_prime))
    (H_sum_diff :
      let diff :=
        let sum :=
          a_prime_c_prime.get_sum [
            local.(KeccakCols.a_prime).[0].[x].[z];
            local.(KeccakCols.a_prime).[1].[x].[z];
            local.(KeccakCols.a_prime).[2].[x].[z];
            local.(KeccakCols.a_prime).[3].[x].[z];
            local.(KeccakCols.a_prime).[4].[x].[z]
          ] in
        sum -F (local.(KeccakCols.c_prime).[x].[z]) in
      diff *F (diff -F 2) *F (diff -F 4) = 0
    ) :
  0 <= x < 5 ->
  0 <= z < 64 ->
  local.(KeccakCols.c_prime).[x].[z] =
  Z.b2z (xorbs [
    Z.odd (local.(KeccakCols.a_prime).[0].[x].[z]);
    Z.odd (local.(KeccakCols.a_prime).[1].[x].[z]);
    Z.odd (local.(KeccakCols.a_prime).[2].[x].[z]);
    Z.odd (local.(KeccakCols.a_prime).[3].[x].[z]);
    Z.odd (local.(KeccakCols.a_prime).[4].[x].[z])
  ]).
```

4 Definition of Keccak

We then define the Keccak algorithm using explicit boolean expressions, following the comments in the Plonky3 code, in the module `ComputeKeccak` of our file `proofs/air.v`. Here is for example our expression of a' with respect to a and c:

```
Definition compute_a_prime
    (a : Array.t (Array.t (Array.t bool 64) 5) 5)
    (c : Array.t (Array.t bool 64) 5) :
    Array.t (Array.t (Array.t bool 64) 5) 5 :=
  {|
    Array.get y := {|
      Array.get x := {|
        Array.get z :=
          xorbs [
            a.[y].[x].[z];
            c.[(x + 4) mod 5].[z];
            c.[(x + 1) mod 5].[(z + 63) mod 64]
          ];
      |}
    |}
  |}.
```

Our definition of Keccak on booleans is of about 120 lines. We have not been able to test it as it is extremely inefficient, due to the use of functions to represent arrays. Our plan is to prove it as equivalent to the Keccak definition we use in the coq-of-solidity project and that it runs successfully on the examples of the Solidity repository.

5 Final verification steps

For the final steps of the verification of Keccak, we show two things:

1. The constraints of the circuit imply the result of a round a''' to be equal to the result of our boolean definition for Keccak. This verification is rather straightforward, as we have already pre-processed the Keccak constraints in a nice format.
2. The successive rounds are connected, so that when we execute 24 rounds in a row we necessarily get the result of Keccak.

We express this last property as follows:

```
Lemma posts_imply {p} `{Prime p} (rows' : Z -> KeccakCols.t)
    (preimages : Z -> Array.t (Array.t (Array.t bool 64) 5) 5) :
  let rows i := M.map_mod (rows' i) in
  ( (* We assume we validated the circuit on all the rows. Note that we
       assume here that we are always transitioning. *)
    forall i, 0 <= i ->
    Post.t (rows i) (rows (i + 1)) (i =? 0) true
  ) ->
  ( (* We assume the preimages are given by the [preimages] function at
       the beginning of each round. *)
    forall i, 0 <= i ->
    i mod NUM_ROUNDS = 0 ->
    forall x, 0 <= x < 5 ->
    forall y, 0 <= y < 5 ->
    forall limb, 0 <= limb < U64_LIMBS ->
    (rows i).(KeccakCols.preimage).[y].[x].[limb] =
    Limbs.of_bools BITS_PER_LIMB
      (Array.get (preimages (i / NUM_ROUNDS)).[y].[x])
      limb
  ) ->
  ( (* We prove that we get the Keccak output in the [a_prime_prime_prime]
       array of the last round. *)
    forall i, 0 <= i ->
    forall x, 0 <= x < 5 ->
    forall y, 0 <= y < 5 ->
    forall limb, 0 <= limb < U64_LIMBS ->
    let final_index := NUM_ROUNDS * (i / NUM_ROUNDS) + 23 in
    Impl_KeccakCols.a_prime_prime_prime (rows final_index) y x limb =
    Limbs.of_bools BITS_PER_LIMB
      (Array.get
        (ComputeKeccak.compute_keccak (preimages (i / NUM_ROUNDS))).[y].[x]
      )
      limb
  ).
```

The important parts are that:

- We represent rows as a function from an index to the content of the row.
- We assume that we have successfully executed the circuits on each successive pair of rows, named `local` and `next` in Plonky3.
- We assume a list of *preimages* to be fed to the circuit at the beginning of each Keccak round.
- From there, we show that the (virtual) column a''' of the final row of a round must be equal to the *preimage* given at the beginning of the row. This proves both the determinism (there can be at most one possible output) and the functional correctness (we are computing the Keccak function).

The core of the proof is about reasoning by induction over the rounds, showing that we correctly propagate the Keccak rounds until we have iterated 24 times.

Most of the Rocq proofs are done using the standard interactive proof mode rather than full automation, but rely on keeping a high-level structure for the constraints to simplify the reasoning (no loops unrolling, for example).

We have also started verifying the *Blake3* implementation in Plonky3 using similar techniques, but we cannot report on it as this is still a work in progress.

6 Findings

We have not found any issues in the safety or correctness of the circuits. We believe that one could remove the `preimage` columns, as these seem to be used to initialize the `a` columns and that can be done directly, saving a few constraints.

7 Conclusion

We thank the Ethereum Foundation for giving us the trust and opportunity to work on this project! We hope it can be applied to the formal verification of other pre-compiles or full zkVMs. We believe this approach scales proportionally to the complexity or quantity of changes in the circuits, as it closely follows the structure of the code.

Formal Land