Report: Formal Specification of Revm

Formal Land

2026-02-16

Formal Land

Contents

1 Summary

Here is the description of the grant targets:

> Month 1: Formalisation of the Revm in idiomatic Rocq code, with tests to compare the Rocq and Rust versions covering each instruction.
>
> Month 2: Proof that the formalization of the Revm is equivalent to the Rust source code as translated by `rocq-of-rust`. Will use two versions Rocq of the Rust code: one that is written by hand and amenable for the proofs; and one that is generated automatically by `rocq-of-rust` that is more verbose and takes into account Rust implementation details, such as pointers, that can be abstracted away for the proofs.

We completed these targets for 94% of the instructions, and are continuing for the remaining ones, which might be completed at the time of reading for this report. Note that we target the *body* of the functions defining the instructions. Their dependencies are axiomatized or verified depending on the case, and would require separated work to be fully formally verified.

We have also shown for all the functions we verified that there are no possible panics, assuming that their dependencies are correctly axiomatized.

The main folder with the functional specification of the instructions, which we call "simulations", together with their proofs of equivalence with the Rust source code, is in `https://github.com/formal-land/rocq-of-rust/tree/main/RocqOfRust/revm/revm_interpreter/instructions/simulate`

The tests for the instructions, making sure the specifications are efficiently computable, a criterion of good quality, are in the folder `https://github.com/formal-land/rocq-of-rust/tree/main/RocqOfRust/revm/revm_interpreter/instructions/tests`

This work took longer than expected, due to unanticipated complexities with the Rust language. As a result, we also improved `rocq-of-rust`, the tool we use to build and verify this functional specification, in order to handle much larger codebases compared to what we were doing before.

In total, we produced about 50,000 lines of Rocq for this project, as well as a hard-to-evaluate number of Rust lines to improve the `rocq-of-rust` tool itself. We did not modify the code of Revm to make our verification, working on the fixed commit `80099a7702084332b8de4a99dabe0095b5cde705` of Revm. This shows that our methodology is applicable to a wide variety of *safe* Rust code. Our tool `rocq-of-rust` is one of the few, if not the only, tools capable of handling such general translations. We do not rely on the borrow checker, so the techniques in use could also be applied to other languages such as Go or C.

We are happy to discuss any details in this report and the project.

2 What we get from it

Having a verified functional specification for the Rust code of Revm is important to:

- Build a formally verified RISC-V bytecode to run Revm on top of a RISC-V zkVM.
- Show the absence of runtime errors in the Revm implementation.
- Show the equivalence of Revm with other EVM implementations.

- Get a precise semantics of the EVM in a formal language, and verify further the Rocq and Lean EVM specifications (the link between Rocq and Lean can be done using the `lean-import` tool).

3 Talks and blog posts

We gave three talks over the course of this project:
- Symposium on Formal Methods for RUST at ICSE 2025 May 2, 2025, Ottawa, Canada
- Rust Paris 2025 June 25, 2025, Paris, France
- FOSDEM 2026 February 1, 2026, Brussels, Belgium

We wrote two blog posts:
- Functional specification of the ADD instruction of the EVM
- Functional correctness of STATIC_CALL in Revm

4 Description

The codebase of Revm, a Rust implementation for the EVM, is structured in a modular way with one function per instruction. Primitive functions, to manipulate the stack or the storage, for example, are given as parameters through the `InterpreterTypes` and `Host` traits. We axiomatize those functions, as well as the `ruint` crate providing definitions for 256-bit unsigned integers. For optimization purposes, many definitions use Rust macros, typically to manipulate the stack, do gas accounting, or handle conversions between different integer sizes.

The instructions are grouped by category, with some functions that are actually helper functions for other instructions. Here is the list of functions in the `instructions` folder, with those with a simulation proven as equivalent to the Rust source code. The tests are covering all the instructions, testing at least one input per instruction. These are simpler to write than the equivalence proofs.

We do not look at the EOF opcodes, as they are not part of the current EVM specification.

- ✓ = fully proven (`Qed`)
- × = admitted (`Admitted`)

4.1 arithmetic/

Rust source: `arithmetic.rs`

Instruction	Status
add	✓
addmod	✓
div	✓
exp	✓
mul	✓
mulmod	✓
rem	✓
sdiv	✓
signextend	✓
smod	✓
sub	✓

4.2 bitwise/

Rust source: `bitwise.rs`

Instruction	Status
bitand	✓
bitor	✓
bitxor	✓
byte	✓
eq	✓
gt	✓
iszero	✓
lt	✓
not	✓
sar	✓
sgt	✓
shl	✓
shr	✓
slt	✓

4.3 block_info/

Rust source: `block_info.rs`

Instruction	Status
basefee	✓
blob_basefee	✓
block_number	✓
chainid	✓
coinbase	✓
difficulty	✓
gaslimit	✓
timestamp	✓

4.4 contract/

Rust source: `contract.rs`

Instruction	Status
call	✓
call_code	✓
delegate_call	✓
extcall_input	✓
static_call	✓

4.5 control/

Rust source: `control.rs`

Instruction	Status
invalid	✓
jump	✓
jump_inner	✓
jumpdest_or_nop	✓
jumpi	✓
pc	✓
ret	✓
return_inner	✓
revert	✓
stop	✓
unknown	✓

4.6 host/

Rust source: `host.rs`

Instruction	Status
balance	✓
blockhash	✓
extcodecopy	✓
extcodehash	✓
extcodesize	✓
log	×
selfdestruct	×
selfbalance	✓
sload	✓
sstore	✓
tload	✓
tstore	✓

4.7 memory/

Rust source: `memory.rs`

Instruction	Status
mcopy	✓
mload	✓
msize	✓
mstore	✓
mstore8	✓

4.8 stack/

Rust source: `stack.rs`

Instruction	Status
dup	✓
pop	✓
push	✓
push0	✓
swap	✓

4.9 system/

Rust source: `system.rs`

Instruction	Status
address	✓
calldatacopy	✓
calldataload	×
calldatasize	✓
caller	✓
callvalue	✓
codecopy	✓
codesize	✓
gas	✓
keccak256	✓
memory_resize	✓
returndatacopy	✓
returndatasize	✓

4.10 tx_info/

Rust source: `tx_info.rs`

Instruction	Status
gasprice	✓

There are two additional instructions in `tx_info`, for which we do not have a translation in Rocq as we do not handle `dyn` types in `rocq-of-rust` yet.

5 Walkthrough

5.1 Steps

At a high-level, what we are doing is to formally verify a translation from the imperative Rust implementation of Revm to an idiomatic and purely functional representation in Rocq. We use our tool `rocq-of-rust` proceeding in the following steps:

1. From Rust to a deep-embedding in Rocq, running `rocq-of-rust`. (fully automated)
2. Adding back the type information and resolving traits, what we name the "link" phase. (semi-automated and greatly helped by AI for the manual part)
3. Defining an idiomatic and purely functional version of the Rust code (manual and partially helped by AI)
4. Proving the equivalence of the two versions (semi-automated and partially helped by AI)

As we continue verifying Rust code, we plan to increase the automation of the process, and we already have several ideas for it.

5.2 Instructions

If we consider the `CALLDATASIZE` opcode, its Rust implementation is:

```
pub fn calldatasize<WIRE: InterpreterTypes, H: Host + ?Sized>(
    interpreter: &mut Interpreter<WIRE>,
    _host: &mut H,
) {
    gas!(interpreter, gas::BASE);
    push!(interpreter, U256::from(interpreter.input.input().len()));
}
```

It takes as parameters two trait implementations for `InterpreterTypes` and `Host` with their corresponding state values `interpreter` and `_host`. It runs the two macros `gas!` and `push!` to update the interpreter state with the expected values.

When translating this code to Rocq through `rocq-of-rust` we obtain a code of 300 lines starting with:

```
 Definition calldatasize
   (ε : list Value.t) (τ : list Ty.t)
   (α : list Value.t) : M :=
   match ε, τ, α with
   | [], [ WIRE; H ], [ interpreter; _host ] =>
     ltac:(M.monadic
       (let interpreter :=
         M.alloc (|
           Ty.apply
             (Ty.path "&mut")
             []
             [ Ty.apply
               (Ty.path
                 "revm_interpreter::interpreter
                   ::Interpreter")
               [] [ WIRE ] ],
           interpreter
         |) in
       let _host :=
         M.alloc (|
           Ty.apply (Ty.path "&mut") [] [ H ],
           _host
         |) in
       M.catch_return (Ty.tuple []) (|
         ltac:(M.monadic
           (M.read (|
             let~ _ : Ty.tuple [] :=
               M.match_operator (|
                 Ty.tuple [],
                 M.alloc (|
                   Ty.tuple [],
                   Value.Tuple []
                 |),
                 [
                   fun γ =>
                     ltac:(M.monadic
                        (let γ :=
                          M.use
                            (M.alloc (|
                              Ty.path "bool",
                              M.call_closure (|
                                Ty.path "bool",
                                UnOp.not,
```

This code is much longer because we unfold the macros, add type annotations from the Rust compiler, and make explicit all the conversions and reference manipulations done by the Rust compiler.

Before showing the equivalence with the idiomatic purely functional representation, we add type information and trait resolution:

```
Instance run_calldatasize
  {WIRE H : Set} `{Link WIRE} `{Link H}
  {WIRE_types : InterpreterTypes.Types.t}
  `{InterpreterTypes.Types.AreLinks WIRE_types}
  (run_InterpreterTypes_for_WIRE :
    InterpreterTypes.Run WIRE WIRE_types)
  (interpreter :
    '&mut (Interpreter.t WIRE WIRE_types))
  (_host : '&mut H) :
  Run.Trait
    instructions.system.calldatasize
    [] [ Φ WIRE; Φ H ]
    [ φ interpreter; φ _host ]
    unit.
Proof.
  constructor.
  run_symbolic.
Defined.
Global Opaque run_calldatasize.
```

This is handled automatically with the `run_symbolic` tactic. Then we define the specification in idiomatic Rocq code:

```
Definition calldatasize
    {WIRE : Set} `{Link WIRE}
    {WIRE_types : InterpreterTypes.Types.t}
    `{InterpreterTypes.Types.AreLinks WIRE_types}
    {IInterpreterTypes :
      InterpreterTypes.C WIRE_types}
    (interpreter :
      Interpreter.t WIRE WIRE_types) :
    Interpreter.t WIRE WIRE_types :=
  gas_macro interpreter constants.BASE id
    (fun interpreter =>
  let input :=
    IInterpreterTypes
      .(InterpreterTypes.InputsTrait_for_Input)
      .(InputTraits.input)
      .(RefStub.projection)
        interpreter.(Interpreter.input) in
  let length : usize :=
    Impl_Slice.len input in
  push_macro interpreter
    (Impl_Uint.from length)
    id id
  ).
```

Note that for short instructions like this one, all the verification steps can be written by AI, once enough other instructions are verified as examples.

Finally, the proof of equivalence between the functional specification and the Rust code is as follows:

```
Lemma calldatasize_eq
    {WIRE H : Set} `{Link WIRE} `{Link H}
    {WIRE_types : InterpreterTypes.Types.t}
    `{InterpreterTypes.Types.AreLinks WIRE_types}
    (run_InterpreterTypes_for_WIRE :
      InterpreterTypes.Run WIRE WIRE_types)
    (IInterpreterTypes :
      InterpreterTypes.C WIRE_types)
    (InterpreterTypesEq :
      InterpreterTypes.Eq.t WIRE WIRE_types
        run_InterpreterTypes_for_WIRE
        IInterpreterTypes)
    (interpreter :
      Interpreter.t WIRE WIRE_types)
    (host : H) :
  let ref_interpreter := make_ref 0 in
  let ref_host := make_ref (A := H) 1 in
    {{
      SimulateM.eval_f
        (run_calldatasize
          run_InterpreterTypes_for_WIRE
          ref_interpreter ref_host)
        [interpreter; host]%stack
      ≃
      (
        Output.Success tt,
        [calldatasize interpreter; host]%stack
      )
    }}.
Proof.
  with_strategy transparent [run_calldatasize]
    unfold calldatasize, run_calldatasize;
    cbn.
  gas_macro_eq idtac.
  s. {
    apply InterpreterTypesEq.
  }
  s. {
    pose proof (Impl_Slice.len_eq (T := u8))
      as H_apply.
    s_apply H_apply.
  }
  s. {
    s_apply Impl_Uint.from_eq.
  }
  push_macro_eq InterpreterTypesEq.
  s.
Qed.
```

This is the main part where we think we can make more automation.

5.3 Traits

Traits are an important feature of Rust and are extensively used in Revm. Here is, for example, the `InterpreterTypes` trait:

```
pub trait InterpreterTypes {
    type Stack: StackTrait;
    type Memory: MemoryTrait;
    type Bytecode: Jumps + Immediates
        + LegacyBytecode + EofData
        + EofContainer + EofCodeInfo;
    type ReturnData: ReturnData;
    type Input: InputsTrait;
    type SubRoutineStack: SubRoutineStack;
    type Control: LoopControl;
    type RuntimeFlag: RuntimeFlag;
    type Extend;
}
```

with the `MemoryTrait` trait:

```
pub trait MemoryTrait {
    fn set_data(
        &mut self,
        memory_offset: usize,
        data_offset: usize,
        len: usize,
        data: &[u8],
    );
    fn set(
        &mut self,
        memory_offset: usize,
        data: &[u8],
    );
    fn size(&self) -> usize;
    fn copy(
        &mut self,
        destination: usize,
        source: usize,
        len: usize,
    );
    fn slice(
        &self, range: Range<usize>,
    ) -> impl Deref<Target = [u8]> + '_;
    fn slice_len(
        &self, offset: usize, len: usize,
    ) -> impl Deref<Target = [u8]> + '_;
    fn resize(
        &mut self, new_size: usize,
    ) -> bool;
}
```

These traits feature various advanced Rust features for the definition of traits. We type these in Rocq in the "link" phase as:

```
Module InterpreterTypes.
  Module Types.
    Record t : Type := {
      Stack : Set;
      Memory : Set;
      Memory_Synthetic : Set;
      Memory_Synthetic1 : Set;
      Bytecode : Set;
      ReturnData : Set;
      Input : Set;
      SubRoutineStack : Set;
      Control : Set;
      RuntimeFlag : Set;
      Extend : Set;
    }.

    Class AreLinks (types : t) : Set := {
      H_Stack :: Link types.(Stack);
      H_Memory :: Link types.(Memory);
      H_Memory_Synthetic ::
        Link types.(Memory_Synthetic);
      H_Memory_Synthetic1 ::
        Link types.(Memory_Synthetic1);
      H_Bytecode :: Link types.(Bytecode);
      H_ReturnData :: Link types.(ReturnData);
      H_Input :: Link types.(Input);
      H_SubRoutineStack ::
        Link types.(SubRoutineStack);
      H_Control :: Link types.(Control);
      H_RuntimeFlag ::
        Link types.(RuntimeFlag);
      H_Extend :: Link types.(Extend);
    }.
  End Types.
  Export (hints) Types.

  Class Run
      (Self : Set) `{Link Self}
      (types : Types.t)
      `{Types.AreLinks types} :
      Set := {
    Stack_IsAssociated :
      IsTraitAssociatedType
        "revm_interpreter::interpreter_types
          ::InterpreterTypes"
        [] [] (Φ Self)
        "Stack" (Φ types.(Types.Stack));
    run_StackTrait_for_Stack ::
      StackTrait.Run types.(Types.Stack);
    ...
  }.
End InterpreterTypes.
Export (hints) InterpreterTypes.
```

and:

```
Module MemoryTrait.
  Definition trait (Self : Set) `{Link Self}
      : TraitHeader.t :=
    {|
      TraitHeader.trait_name :=
      "revm_interpreter::interpreter_types
        ::MemoryTrait";
    TraitHeader.trait_consts := [];
    TraitHeader.trait_tys := [];
    TraitHeader.self_ty := Φ Self;
  |}.

  Class Method_set_data (Self : Set)
      `{Link Self} : Set := {
    set_data : PolymorphicFunction.t;
    set_data_is_method ::
      IsTraitMethod.C (trait Self)
        "set_data" set_data;
    run_set_data
      (self : '&mut Self)
      (memory_offset data_offset len : usize)
      (data : '& (list u8)) ::
        Run.Trait set_data [] []
          [ φ self; φ memory_offset;
            φ data_offset;
            φ len; φ data ]
          unit;
  }.

  ...

  Class Run
      (Self Synthetic Synthetic1 : Set)
      `{Link Self} `{Link Synthetic}
      `{Link Synthetic1} :
      Set := {
    method_set_data ::
      Method_set_data Self;
    method_set :: Method_set Self;
    method_size :: Method_size Self;
    method_copy :: Method_copy Self;
    ...
  }.
End MemoryTrait.
Export (hints) MemoryTrait.
```

The code above is quite verbose, but mostly generated by AI and verified as being correct later in the proofs. We could further automate its generation for common cases. We also give a functional specification for these traits:

```
Module InterpreterTypes.
  Class C
      (WIRE_types :
        InterpreterTypes.Types.t)
      `{InterpreterTypes.Types.AreLinks
          WIRE_types} :
      Type := {
    (* type Stack: StackTrait; *)
    StackTrait_for_Stack ::
      StackTrait.C
        WIRE_types
          .(InterpreterTypes.Types.Stack);
    (* type Memory: MemoryTrait; *)
    MemoryTrait_for_Memory ::
      MemoryTrait.C
        WIRE_types
          .(InterpreterTypes.Types.Memory)
        WIRE_types
          .(InterpreterTypes.Types
              .Memory_Synthetic)
        WIRE_types
          .(InterpreterTypes.Types
              .Memory_Synthetic1);
    ...
  }.
End InterpreterTypes.
Export (hints) InterpreterTypes.
```

and:

```
Module MemoryTrait.
  Class C
      (Self Synthetic Synthetic1 : Set)
      `{Link Self} `{Link Synthetic}
      `{Link Synthetic1} : Set := {
    set_data (self : Self)
      (memory_offset data_offset len : usize)
      (data : list u8) : Self;
    set (self : Self)
      (memory_offset : usize)
      (data : list u8) : Self;
    size (self : Self) : usize;
    copy (self : Self)
      (destination source len : usize) : Self;
    slice (self : Self)
      (range : Range.t usize) : Synthetic;
    Deref_for_Synthetic ::
      Deref.C Synthetic (list u8);
    slice_len (self : Self)
      (offset len : usize) : Synthetic1;
    Deref_for_Synthetic1 ::
      Deref.C Synthetic1 (list u8);
    resize (self : Self) (new_size : usize)
      : bool * Self;
  }.
End MemoryTrait.
Export (hints) MemoryTrait.
```

6 Potential future directions

There are several directions that we think can be worked on in the future, with the aim to bring more security to the (zk)EVM platforms:

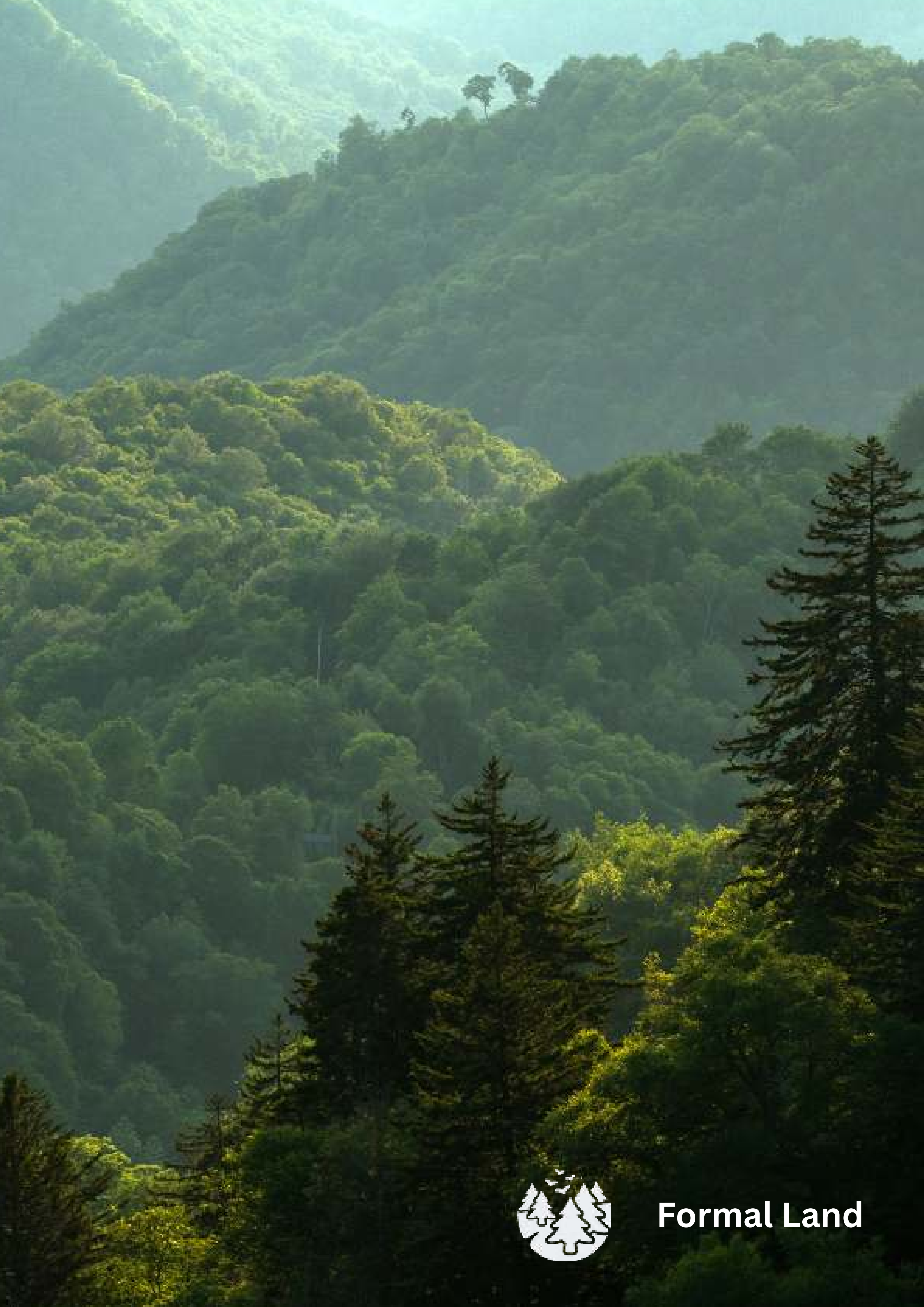
1. Extending the scope of this verification work to recursively cover more of the Rust dependencies, and hence reduce the trusted codebase.
2. Maintain this functional specification so that it is up-to-date with the latest commit of the Revm repository.
3. Prove the equivalence with other EVM imperative implementations, such as Geth or the Python Specs, following the same approach as we have done with the Rust language.
4. Prove the equivalence with the Rocq and Lean EVM specifications that exist. This

can be followed by a formally verified compilation of these implementations through projects like Peregrine. Combining both, this offers a way to compile Revm down to RISC-V in a formally verified way.

5. Improve the `rocq-of-rust` tool so that other projects can use it more easily, maybe even without our help.
6. Work on pushing the equivalence proofs through the Rust compiler, down to the assembly language. There are various ways of doing this, with different constraints, but the general goal is to compile Revm or other Rust projects in a verified way, with the same or similar speed as with the Rust compiler. This could be combined with CompCert to avoid verifying the LLVM pipeline at some cost of performance (likely a 2x cost).

7 Thanks

We thank the Ethereum Foundation for the trust and the grant for this project.


Formal Land