Isabelle/EVM: A Novel Formalization of the EVM in Isabelle/HOL

Ashley Card
Department of Computer Science, University of Exeter, UK

Diego Marmsoler
Department of Computer Science, University of Exeter, UK

Abstract

Smart contracts deployed on the Ethereum blockchain execute on the Ethereum Virtual Machine (EVM) and handle financial operations such as payments, asset transfers, and auctions. Given the high value they control, correctness in these contracts is critical, as errors and vulnerabilities have led to losses totalling hundreds of millions of dollars. To address this problem, we develop a novel formalization of the EVM. Compared to existing formalizations, our formalization is in Isabelle/HOL, covers all current EVM opcodes, and formalizes cross-contract execution. Thus, it allows us to express properties which are out of scope for other formalizations. To allow for the execution of our formalization, we implement a code generator, allowing it to be exported as a stand-alone Haskell program. We then validate the semantics by executing 25 000 test cases from the official Ethereum test suite. Our formalization can be used to verify concrete smart contracts but also to reason about the correctness of tools and techniques which manipulate bytecode, such as compilers or optimizers.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Ethereum Virtual Machine, Isabelle/HOL, Smart Contracts

Digital Object Identifier 10.4230/OASIcs.FMBC.2026.3

Supplementary Material *Software (Source Code)*: `https://github.com/ashleycard/Isabelle-EVM` archived at `swh:1:dir:18e0a00cb1f62f425c210f6876600b7ac43e5673`

Funding This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

1 Introduction

The Ethereum blockchain executes programs called smart contracts through the Ethereum Virtual Machine (EVM), a stack-based virtual machine that processes bytecode instructions. The EVM serves as the execution layer for Ethereum and EVM-compatible blockchains such as Polygon[1], running smart contracts that manage billions of dollars in assets and supporting high-volume decentralized finance applications [1, 13]. Developers usually write smart contracts in high-level languages such as Solidity or Vyper. These contracts are then compiled into EVM bytecode, which consists of the low-level instructions that the Ethereum Virtual Machine runs.

Since smart contracts are often used to automate financial transactions, bugs in them are particularly costly, as evidenced by substantial financial losses from high-profile vulnerabilities. The Balancer hack, for example, resulted in over USD 100 million in stolen funds [14], despite being one of the most extensively audited protocols in DeFi. The Vyper compiler bug in 2023 led to USD 52 million in stolen funds [10]; this was a genuine compiler-level bug in which

[1] `https://polygon.technology/`

© Ashley Card and Diego Marmsoler;
licensed under Creative Commons License CC-BY 4.0

reentrancy protection mechanisms failed during compilation, producing incorrect bytecode from correct high-level code. This creates a strong case for formal verification of smart contracts as well as tools which manipulate them, such as compilers and static analysers.

To enable this, with this paper we provide a novel formalization of the EVM for the Isabelle/HOL proof assistant [21]. Our formalization differs from existing formalizations in three ways:

- It is formalized in Isabelle/HOL. Thus it complements existing formalizations for other proof assistants.
- It covers all current EVM opcodes. Thus we can use it to verify EVM programs which include more recent opcodes.
- It formalizes cross-contract execution which allows to formalize properties which are out-of-scope for other formalizations.

Our formalization is executable and to make it more efficient we also implement a code-generator for Haskell. This allows us to export a concrete EVM program to Haskell and simulate an execution. To evaluate our formalization we test it against the official Ethereum test suite[2]. To this end we execute 25 000 test cases. This shows that our formalization is faithful to the semantics of the EVM and verification results are indeed trustworthy. Our formalization as well as the testing framework are available online at `https://github.com/ashleycard/Isabelle-EVM`.

The remainder of the paper is organized as follows. Section 2 provides an overview of the Ethereum Virtual Machine and Section 3 details our formalization. In particular, Subsection 3.1 describes how we represent EVM instructions and how we model EVM states, Subsection 3.2 describes how we formalize instruction execution and Subsection 3.3 explains the code generation setup. Section 4 describes how we evaluated our formalization and Section 5 discusses related work. Finally, in Section 6 we summarize our work and discuss some important findings in more detail.

2 Overview of the Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) [23] is a deterministic, stack-based virtual machine that executes smart contract bytecode, most commonly compiled from Solidity, and manages the global state of the Ethereum blockchain. It provides the execution environment in which contracts run, covering computation, storage, and interactions between accounts.

The global state of the EVM is the persistent state of the Ethereum blockchain at a given point in time. It is a partial mapping from addresses to accounts. Accounts come in two types: externally owned accounts (EOAs) and contract accounts. EOAs are controlled by private keys held by users, while contract accounts are controlled by their code. All accounts have a **balance**, the amount of Ether held, and a **nonce**, a counter that increments with each transaction from the account. Contract accounts additionally have **code**, the executable bytecode, and **storage**, a persistent key-value store.

Transactions originate from EOAs and serve as the entry point to EVM executions. A transaction may deploy a new contract or target an existing account. If the target is a user account, only balances are affected; if it targets a contract account, the EVM executes the associated bytecode. Each transaction specifies a gas limit that sets the maximum computational effort available for execution, preventing infinite loops by terminating execution when the limit is reached.

[2] `https://github.com/ethereum/tests`

Within a transaction, contracts can interact with other accounts using CALL-type instructions. When a contract calls another contract, a new execution context is created with its own stack, memory, program counter, and allocated gas. By contrast, calls to externally owned accounts only transfer funds and do not create a new execution context. Contract-to-contract calls can be nested up to a maximum depth of 1024, with gas passed from the caller to the callee for each nested call. If a nested call fails, all of its state changes are reverted, but execution of the outer call continues with the remaining gas.

Contract execution is deterministic: given the same global state, input data, and gas, running the same code always produces identical results. Each instruction manipulates the execution state by pushing and popping stack values, reading and writing memory, or updating storage. Execution terminates when the code completes, when the gas runs out, or when an error occurs. In all failure cases, all state changes from that execution context are rolled back, except for the gas already consumed. This includes both involuntary failures such as gas exhaustion, and deliberate reverts, where the contract itself executes a `REVERT` instruction to abort execution.

As a simple example, consider Listing 1, which shows a Solidity function that adds two unsigned integers.

Listing 1: Solidity function for `add`.

```
function add(uint a, uint b) returns (uint) {
  return a + b;
}
```

The function shown in Listing 1 compiles to a sequence of EVM instructions as shown in Listing 2.

Listing 2: EVM assembly for `add`.

```
PUSH1 0x00      // Push 0 onto stack (byte offset in calldata)
CALLDATALOAD    // Load 32 bytes from calldata at offset 0 (argument a)
PUSH1 0x20      // Push 32 onto stack (next byte offset)
CALLDATALOAD    // Load 32 bytes from calldata at offset 32 (argument b)
ADD             // Pop two values, push their sum
PUSH1 0x00      // Push 0 onto stack (memory position)
MSTORE          // Store result in memory at position 0
PUSH1 0x20      // Push 32 (size of return data in bytes)
PUSH1 0x00      // Push 0 (memory position to read from)
RETURN          // Return 32 bytes from memory position 0
```

3 Isabelle/EVM

Isabelle/EVM is an executable formalization of the Ethereum Virtual Machine in Isabelle/HOL. It provides the full EVM semantics, including cross-contract execution, and covers the Shanghai[3] and Cancun[4] forks. The formalization is designed to be easily extensible, and a stand-alone executable implementation can be generated, enabling validation against the Ethereum test suite.

[3] https://blog.ethereum.org/2023/03/28/shapella-mainnet-announcement
[4] https://blog.ethereum.org/2024/02/27/dencun-mainnet-announcement

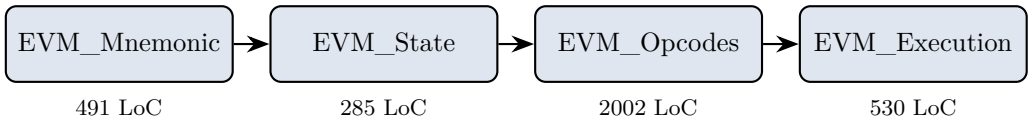

Figure 1 Isabelle/EVM theory structure.

Isabelle/EVM is available online[5] and its file structure is depicted in Figure 1. Theory `EVM_Mnemonic.thy` defines instruction representations, `EVM_State.thy` specifies the state structure, including accounts and environment, `EVM_Opcodes.thy` encodes the semantics and gas costs of individual instructions, and `EVM_Execution.thy` implements overall transaction execution. The lines of code for each component are shown below each box. In total the formalization comprises 3 308 lines of code.

In the following we are going to describe Isabelle/EVM in more detail. Its description is organized into three subsections. *State Formalization* describes how the EVM state is constructed, including accounts, execution state, the environment, instructions, and state transitions. *Transaction Execution* covers how transactions are executed, including opcode behaviour and gas costs. *Code Generation* explains how a stand-alone executable implementation can be produced for validation, including some optimizations to improve performance.

3.1 State Formalization

Our formalization models the execution of a transaction. As shown in Figure 2, this includes contract interactions, call and returns, and global state management across all execution contexts within the transaction boundary.

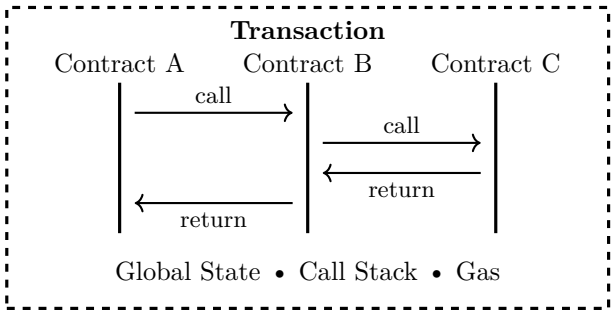

Figure 2 Transaction context.

3.1.1 Instruction Representation

We represent EVM bytecode using an assembly-like language, where each instruction corresponds to a single opcode. To simplify the formalization, instructions are grouped into categories and then combined into a single mnemonic datatype.

`Push0` is defined as a separate constructor as it corresponds to a dedicated opcode that pushes zero onto the stack without an explicit operand, unlike `Push` which encodes both a size and a value. The `Push` constructor takes a `256 word` value, as 256 bits corresponds to

[5] `https://github.com/ashleycard/Isabelle-EVM`

Listing 3: EVM Mnemonic Datatype.

```
datatype arith_op =
  Stop | Add | Mul | Sub | Div | Sdiv | Mod | Smod | Addmod | Mulmod |
  Exp | Signextend

datatype push_op = Push0 | Push push_size "256 word"

datatype mnemonic =
  Arithmetic arith_op | Comparison comparison_op | Bitwise bitwise_op |
  EnvInfo env_info_op | BlockInfo block_op | StackOp stack_op |
  PushOp push_op | DupOp dup_op | SwapOp swap_op |
  LogOp log_op | SystemOp system_op | ControlFlow control_op
```

the maximum push size of 32 bytes. As shown in Listing 3 we support 12 main categories of instructions. *Arithmetic* captures the basic arithmetic operators such as addition and multiplication (e.g., `ADD`, `MUL`). The supported operations for this category are also shown in Listing 3. *Comparison* captures basic comparison operators such as less than or greater than (e.g., `LT`, `GT`); *Bitwise* covers bit-level and logical operations (e.g., `AND`, `NOT`); *EnvInfo* and *BlockInfo* capture access to environmental and block-related information (e.g., `BALANCE`, `BLOCKHASH`); *StackOp*, *PushOp*, *DupOp*, and *SwapOp* cover stack manipulation operations (e.g., `POP`, `PUSH1`, `DUP1`, `SWAP1`); *LogOp* represents event logging (e.g., `LOG1`); *SystemOp* captures system-level interactions (e.g., `CALL`, `CREATE`); and *ControlFlow* models control-flow altering instructions (e.g., `JUMP`).

For convenience, we provide syntactic sugar using Isabelle's syntax translations, allowing an opcode such as `ADD` to be parsed as `Arithmetic Add`. In addition, we simplify the representation of `PUSH` opcodes: rather than defining 32 separate variants, we use a single parametric definition with a custom type that restricts push sizes to 1-32 bytes.

Programs in our formalization are represented at two levels: as mnemonics and as bytecode. Each mnemonic has a corresponding semantic function that defines its effect on the state. Programs are written using mnemonics for readability, and the `compile` function produces the corresponding bytecode used for execution. This translation is a straightforward one-to-one mapping, introducing no complexity that could itself be a source of errors. Listing 4 shows an example Isabelle/EVM program in Isabelle.

Listing 4: Example Isabelle/EVM Program.

```
definition add_prog where
  "add_prog = compile [
    PUSH1 1,              (* 0x60 0x01 | Push 1 onto stack [1] *)
    PUSH1 1,              (* 0x60 0x01 | Push 1 onto stack [1,1] *)
    ADD                   (* 0x01 | Add the top two elements [2] *)
  ]"                      (* Bytecode: 0x6001600101 *)
```

3.1.2 Accounts and Execution State

The `evm_state` record shown in Listing 5 represents the *mutable* machine state during execution. It tracks the stack, memory, accounts, program counter, gas, and other metadata not shown. Each `account` contains a balance, persistent and transient storage, an optional code field, and a nonce. Type `evm_word` denotes 256-bit words, `storage` is a mapping from 256-bit words to 256-bit words, and `address` represents 160-bit words. The `Accounts` field

is a partial mapping from addresses to accounts; if an address is absent, it is treated as an empty account with zero balance, zero nonce, empty storage, and no code. The `Stack` is a list of 256-bit words, while `Memory` is represented as a finite map from byte addresses to 8-bit words, with a `Memory_size` field tracking the highest accessed address. A finite map was chosen over a list as sparse, high-indexed memory accesses would make a list very inefficient. Some field explanations are given inline as comments.

Listing 5: EVM State.

```
record account =
  Balance :: evm_word          (* account balance *)
  Storage :: storage           (* persistent storage *)
  Transient_storage :: storage (* cleared at end of transaction *)
  Code :: "code option"        (* contract code, or None for EOAs *)
  Nonce :: evm_word            (* number of transactions / creations
                                  performed by the account *)
record evm_state =
  Stack :: stack               (* stack of 256-bit words *)
  Memory :: memory             (* byte-addressable memory *)
  Memory_size :: nat           (* highest accessed memory address in bytes *)
  Accounts :: "address ⇀ account"
    (* partial mapping; missing entries are treated as empty accounts *)
  Accessed_addresses :: accessed_addresses
    (* accounts touched during execution, relevant for gas costs *)
  Accessed_storage_keys :: accessed_storage_keys
    (* storage locations accessed, also influencing gas costs *)
  Pc :: nat                     (* program counter *)
  Gas :: gas                    (* remaining gas *)
  Return_data :: return_data    (* return buffer from calls *)
  Halted :: bool                (* normal termination flag *)
  Error :: bool                 (* error flag *)
  Call_depth :: nat             (* current call depth *)
  ...
```

A simplified example of a state midway through execution is shown in Figure 3. There are two contract accounts, one with address 9 and another with address 10. The code held by each account, together with its balance, storage, and nonce, is shown. Account 9, for example, has a value at storage slot 128, which is reflected in the accessed storage keys as the pair (9, 128). While `Storage` holds the actual persistent data of an account, `Accessed_storage_keys` tracks which storage slots have been touched during the current transaction, used to determine gas costs for storage access. The call depth is 1, indicating that the current state is inside a call, with the program counter at index 5 of the executing code.

3.1.3 Execution Environment

The `evm_env` record, shown in Listing 6, specifies the execution environment in which the EVM operates. It contains *immutable* information about the transaction, the block, and the call context, which is queried by various opcodes. Field descriptions are given inline as comments in the definition below. Certain block-related environment fields are omitted.

As of now we support two different gas schedules, Shanghai and Cancun. Thus, the schedule type is defined as: `datatype schedule = SHANGHAI | CANCUN`. We derive linorder for this type, which provides a default ordering so that comparisons such as `SHANGHAI < CANCUN`

```
"⦇evm_state_example.Stack = [], Memory = [],
    Accounts =
      AList_Mapping.Mapping
       [(9, ⦇account.Balance = 0, Storage = AList_Mapping.Mapping [(128, 0)],
              Transient_storage = AList_Mapping.Mapping [],
              Code = Some (IArray [96, 0, 96, 0, 96, 0, 96, 0, 96, 4, 96, 10, 90, 244]), Nonce = 0⦈),
        (10,
         ⦇account.Balance = 0, Storage = AList_Mapping.Mapping [],
            Transient_storage = AList_Mapping.Mapping [], Code = Some (IArray [96, 0, 96, 128, 85, 0]),
            Nonce = 0⦈)],
    Accessed_addresses = {}, Accessed_storage_keys = Abs_fset {(9, 128)}, Pc = 5, Gas = 78724340,
    Return_data = [], Halted = False, Error = False, Call_depth = 1⦈"
  :: "evm_state_example"
```

Figure 3 An example EVM state displayed by Isabelle's `value` command.

Listing 6: EVM Environment.

```
record evm_env =
  Origin_address   :: address    (* account that initiated the transaction *)
  Caller_address   :: address    (* account that made the current call *)
  Current_address  :: address    (* currently executing contract account *)
  Executing_code   :: code       (* code currently executing *)
  Call_value       :: evm_word   (* Ether value sent with the call *)
  Call_data        :: call_data  (* input data supplied to the call *)
  Static_call      :: bool       (* disables state changes in static calls *)
  Gas_price        :: evm_word   (* gas price for this transaction *)
  Gas_limit        :: evm_word   (* transaction gas limit *)
  Schedule         :: schedule   (* gas schedule *)
  ...
```

are valid. This ordering allows conditions to be expressed as range comparisons over schedules, such as "all schedules from Cancun onwards", meaning that supporting a new hard fork requires only inserting a new constructor at the appropriate position in the ordering.

Listing 7 shows an example environment record, with some fields omitted for simplicity. The environment captures the immutable context in which the EVM executes a transaction. It includes addresses such as the origin, caller, and current contract, as well as execution parameters like call value, call data, gas price, gas limit, and the schedule. These fields are separate from the mutable state and do not change during execution, making it easier to reason about programs formally.

Listing 7: Example EVM Environment.

```
definition example_env :: evm_env where
  "example_env ≡ ⦇
     Origin_address = 0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b,
     Caller_address = 0xcccccccccccccccccccccccccccccccccccccccc,
     Current_address = 0x0000000000000000000000000000000000001000,
     Executing_code = IArray [96, 0, 96, 128, 85, 0],
     Call_value = 0x01,
     Call_data = [0x61, 0x00, 0x01, 0x50],
     Static_call = False,
     Gas_price = 0x0a,
     Gas_limit = 0x04c4b400,
     Schedule = CANCUN,
     ...
   ⦈"
```

3.1.4 State Transitions

State transitions in the EVM describe how instructions update control flow, including calls to other contracts and the creation of new accounts. Shown in Listing 8, the `call_type` datatype distinguishes the different kinds of calls, while the `call_frame` record stores a saved frame of execution to which control can return to after a call.

Listing 8: Call and Result Types.

```
datatype call_type = CallOp | StaticCallOp | CallCodeOp address
                   | DelegateCallOp address | CreateOp address

record call_frame =
  Saved_state :: evm_state
  Saved_env :: evm_env
  Return_offset :: nat
  Return_size :: nat
  Call_type :: call_type

datatype exec_result =
    Continue evm_state
  | CallExit evm_state (* return, revert, halt, error results *)
  | Call evm_env evm_state call_frame (* call/create with frame to save *)

type_synonym instr = "evm_env ⇒ evm_state ⇒ exec_result"
```

The possible outcomes of executing an instruction are described by the `exec_result` datatype. Execution may continue in the current context (`Continue`), trigger a call (`Call`), which also covers contract creation, or terminate the current execution (`CallExit`), which encompasses return, revert, halt, and error outcomes. When a `Call` result is produced, the current execution frame is saved and a new execution context is entered; upon completion of the nested call, the saved frame is restored and execution resumes in the outer context. Finally, an `instr` is a function from the environment and current state to an `exec_result`, and this is the type for all EVM opcodes. Having defined the state, we proceed to formalize how transactions are executed.

3.2 Formalizing Transaction Execution

The EVM executes transactions by processing bytecode instructions sequentially, with each operation consuming gas and potentially modifying the state. We formalize this process by first defining the semantics and gas costs of individual opcodes, then specifying how these operations are executed within the transaction context.

3.2.1 Opcodes

The formalization covers around 150 opcodes, spanning the full Cancun instruction set. Each opcode described in Sect. 3.1.1 is formalized as a state transition function that consumes gas, validates stack conditions, and updates the state accordingly. Consider, for example, the formalization of the `ADD` operation shown in Listing 9.

This definition follows a common pattern: gas is consumed first via `consume_gas` (which will be explained in more detail later on), then the stack is pattern-matched to ensure enough operands are available, with stack underflow resulting in an error state. Operations that grow the stack such as `ADDRESS` first check the stack remains below the 1024 element limit.

Listing 9: ADD Opcode.

```
fun ADD_op :: instr where
  "ADD_op env σ = consume_gas ADD env σ (λ σ '.
    case Stack σ' of
      x # y # stk' ⇒ Continue (σ'⦇ Stack := (x + y) # stk' ⦈)
    | _ ⇒ CallExit (σ'⦇ Error := True ⦈))"
```

The formalization of the KECCAK opcode was particularly challenging since we do not have a working formalization of the actual keccak hash function. Thus, we used an abstract definition via a locale, shown in Listing 10.

Listing 10: Keccak Locale.

```
locale keccak256 =
  fixes keccak256 :: "8 word list ⇒ 256 word"

fun KECCAK256_op :: instr where
  "KECCAK256_op env σ =
     consume_gas KECCAK256 env σ (λ σ '.
       case Stack σ' of
         mem_offset # mem_size # stk' ⇒
           let
             ( σ '', bytes) = mem_read (unat mem_offset) (unat mem_size) σ';
             hash = keccak256 bytes
           in
             Continue (σ ''⦇ Stack := hash # stk' ⦈)
       | _ ⇒ CallExit (σ'⦇ Error := True ⦈))"
```

The mem_read function returns the bytes at a given offset and length, along with an updated state that expands memory if necessary, as reading from an uninitialized memory region causes a memory expansion. Then, keccak256 creates a hash from the list of bytes, and this is pushed onto the stack. This abstract definition is later instantiated with a concrete Haskell implementation during code extraction so that it is executable.

The CREATE and CREATE2 opcodes, which rely also on the keccak hash function to compute contract addresses, also use this abstract definition.

3.2.2 Gas Costs

Gas consumption is modelled by two functions shown in Listing 11. The gas_exec function computes the gas cost for each opcode, while consume_gas deducts the cost from the current gas and continues execution or raises an out-of-gas error. The available function checks whether the opcode is available in the given gas schedule.

The gas costs follow the Ethereum specification, with opcodes categorized into different cost tiers such as G_zero for operations like STOP and G_verylow for simple arithmetic. More complex operations, such as memory or storage access, require custom gas cost functions that depend on the current state and environment. For example, the BALANCE opcode has a base cost of G_balance plus an additional address_access_cost that varies depending on whether the target address is already in the accessed addresses set. The definition of the corresponding gas cost function is shown in Listing 12. The case returning 0 has no effect in practice, as a malformed stack triggers an exceptional halt that consumes all remaining gas in the current context regardless. Another dynamic gas cost opcode is the SSTORE opcode, whose

Listing 11: EVM Gas Cost Functions.

```
definition G_verylow where "G_verylow ≡ 3"

fun gas_exec where
  "gas_exec STOP env σ = G_zero" |
  "gas_exec ADD env σ = G_verylow" |
  "gas_exec BALANCE env σ= gas_BALANCE env σ" |
    ...

fun consume_gas where
  "consume_gas mn env σ result =
    (case available (Schedule env) mn of
       False ⇒ CallExit (σ⦇ Error := True ⦈)
     | True ⇒
       let cost = gas_exec mn env σ in
       if Gas σ ≥ cost then result (σ⦇ Gas := Gas σ - cost ⦈)
                       else CallExit (σ⦇ Error := True ⦈))"
```

Listing 12: BALANCE Gas Cost Function.

```
fun gas_BALANCE :: gas_instr where
  "gas_BALANCE env σ =
    case Stack σ of
      addr_word # _ ⇒
        G_balance + address_access_cost σ (ucast addr_word)
    | _ ⇒ 0"
```

cost depends on whether the target storage key exists within the `Accessed_storage_keys` set. In addition, if a storage slot is reset to its original value, `SSTORE` triggers a gas refund which is accumulated in a refund counter in the state. This counter is applied on successful completion of the transaction.

3.2.3 Execution

Transaction execution is implemented through a step function that processes individual instructions and an execution loop that repeatedly calls the step function until termination. The step function fetches the current instruction, executes it, and returns the updated state or signals for calls, returns, or errors. The step function code is omitted for brevity.

The execution loop shown in Listing 13 handles the different result types from `step`. A `Continue` result causes execution to advance to the next instruction, while a `Call` result pushes a new frame onto the call stack to handle contract interactions, including contract creation. The `CallExit` result either pops the call stack to restore the previous execution context or terminates the entire transaction if the call stack is empty. The execution loop is guaranteed to terminate because each step consumes gas, and execution halts when insufficient gas remains for the next operation.

At the end of a transaction, the `top_level_cleanup` function is called to finalise the state and perform necessary clean-up, such as clearing transient storage and applying gas refunds.

The `exec_loop` function is quite complex and Isabelle is not able to automatically verify that it always terminates. While we could still work with `exec_loop` without proving termination, reasoning would be more difficult and we would not be able to generate code

Listing 13: Execution Loop Function.

```
function exec_loop where
 "exec_loop env σ call_stack =
   (if Halted σ ∨ Error σ then σ
    else
      case step env σ of
        Continue σ' ⇒
          exec_loop env σ' call_stack
      | Call new_env new_state frame ⇒
          exec_loop new_env new_state (frame # call_stack)
      | CallExit exit_state ⇒
          (case call_stack of
             [] ⇒ top_level_cleanup env exit_state
           | frame # rest_stack ⇒
               let
                 success = ¬Error exit_state;
                 restored_state = restore_caller_state (Saved_state frame)
                                  frame exit_state success
               in exec_loop (Saved_env frame) restored_state rest_stack))"
```

for it. Thus, we manually proved termination of the function to turn it into a total function. The termination proof for the execution function uses a measure function consisting of two components:

1. The total gas available (sum of the current gas and the gas stored in all call frames)
2. The call stack depth

Operations that continue within the current call frame or create new call frames must strictly decrease the total gas available. Operations that return from a call frame need only ensure gas does not increase, since the reduction in call stack depth alone guarantees progress. This captures both sources of EVM termination: gas consumption within frames and bounded recursion depth.

3.3 Code Generation

Isabelle's code generator [15] can automatically produce code in several target languages, including Haskell, OCaml, Scala and SML. We make use of the Haskell target to extract our EVM formalization to standalone Haskell code, which can be compiled and executed independently of Isabelle. While Isabelle/HOL provides built-in execution mechanisms such as the `value` command, these are not suitable for running large test suites due to performance limitations. The generated code is not always efficient out of the box; however it can be tuned by selecting appropriate types and providing custom code equations. We chose appropriate types for performance in our formalization, such as using an `IArray` (immutable array) instead of a list to represent the EVM code, allowing for efficient access. Another optimization involves importing the red-black tree mapping library in `EVM_Export.thy`, the Isabelle file responsible for Haskell code generation, allowing Isabelle proofs to still use the standard map abstraction while generating efficient code. The file structure for testing is shown in Figure 4: `EVM_Export.thy` specifies the Haskell extraction of the formalization, instantiating abstract operations such as KECCAK using the cryptohash library[6], and

[6] `https://hackage.haskell.org/package/cryptohash`

produces `EVMRuntime.hs`. The resulting Haskell file is then used by a test harness that feeds test cases from the Ethereum test suite into the execution function and compares the resulting state against the expected output. This allows us to test the formalization, and we report the results in the next section.

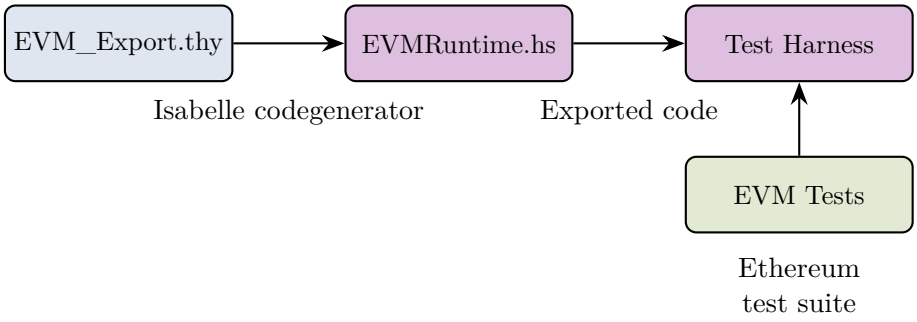

Figure 4 Code generation and testing pipeline.

4 Evaluation

We validated the formalization against the official EVM test suite [12] using the generated Haskell code. For this, we implemented a testing tool that automatically executes the Ethereum test cases and reports the results. This ensures our semantics conforms to the Ethereum specification [23] and Ethereum Improvement Proposals (EIPs), such as EIP-2930 [6]. Each test specifies an initial pre-state and an expected post-state; the tool feeds the pre-state into the generated Haskell code, executes it, and compares the resulting post-state against the expected output. As we only formalized two forks (Cancun, Shanghai), we only use the tests for those.

Our formalization and test system are available at `https://github.com/ashleycard/Isabelle-EVM`. The test suite contains 25 871 unique tests across 2 793 files. Of these files, 2 669 were run while 124 were skipped because they were malformed or timed out. Of the 2 669 test files that we ran, 703 failed and 1 966 passed (74%). Among the 703 failed tests, at least 440 were due to unimplemented precompiled contracts. Precompiled contracts are a set of built-in contracts available at fixed addresses (`0x1`–`0x0a` as of Cancun) that implement computationally intensive operations natively at the client level, rather than as EVM bytecode. Some would be straightforward to add, such as modular exponentiation (`0x5`), while others such as `ecrecover` (`0x1`) require formalizing elliptic curve arithmetic. Consequently, implementing them was out of scope for this formalization. Following the same approach taken for Keccak, precompiled contracts could be introduced as abstract functions with a concrete implementation supplied for testing in Haskell. The remaining (few) failures involve the CREATE opcode and interactions between multiple contracts. Diagnosing these failures was difficult as the testing framework only compares the final post-state against the expected output, giving little insight into where execution went wrong. A more informative approach would be to use a reference Ethereum client to produce a full execution trace, and compare this step-by-step against a trace generated from the Isabelle semantics, to identify where the two differ.

5 Related Work

While tools exist for verifying Solidity source code, such as Isabelle/Solidity [2, 18, 19], SolidiKeY [3], and a Dafny-based verifier by Cassez et al. [9], such guarantees are only meaningful if the compiled bytecode faithfully preserves the verified properties. Without a formal specification of the EVM and its bytecode semantics, there is no formal target against which to verify compilation, leaving a critical gap in the verification chain. A formal EVM specification is therefore essential not only for reasoning directly about bytecode, but also as a foundation for proving compiler correctness and establishing end-to-end guarantees from source code to execution. Certora's Prover [11], the most widely used industrial tool in this space, addresses this in practice through symbolic execution and SMT solving, though it relies on a trusted mapping from Solidity to EVM bytecode, rather than a formally verified one.

Several formalizations of the Ethereum Virtual Machine (EVM) have been developed across different verification frameworks, showing an interest in providing executable and machine checked semantics for Ethereum bytecode. KEVM [16] provides the most extensive specification to date, offering a full operational semantics of the EVM in the K framework [22]. It covers all instructions and yields an executable model suitable for testing and symbolic reasoning. While it represents the most comprehensive effort to date, reasoning in K relies on rewriting semantics and differs from proof assistant based approaches.

Dafny EVM [8] provides a complete formal semantics of the EVM in Dafny, defining instructions as pure functions with explicit preconditions and postconditions. ByteSpector [7] builds upon this formalization to construct formally verified control flow graphs from EVM bytecode, automatically generating Dafny proof objects that encode bytecode semantics. These tools demonstrate how a formal EVM specification can serve as a foundation for practical verification tools. Other formalizations exist in Lean [20] and Coq [4]. The Lean formalization covers both the EVM and Yul, the intermediate representation between Solidity and EVM bytecode. It also establishes a partial equivalence with KEVM, though only in one direction: if KEVM computes a result, the Lean model agrees, but the converse remains unproven. With this work we contribute to this line of research by providing an alternative formalization for the Isabelle proof assistant. This allows analyses to be performed in Isabelle and build upon other verification results in this domain (for example Isabelle/Solidity [19])

There is, however, one work in which the EVM is formalized in Isabelle. A formalization of the EVM was given by Hirai [17], who specified the semantics in Lem, a language whose definitions can be exported to Isabelle/HOL, and was later extended by Amani et al. [5] in Isabelle. The formalization is limited to single-contract execution, with effects of external calls abstracted in the environment. They also introduce a program logic covering a subset of 36 instructions, with the program semantics expressed through a control-flow graph. Using their formalization, they verified an escrow agreement smart contract implemented in Solidity, demonstrating their approach can reason directly about compiled EVM bytecode. There are, however, two main differences to our work. First, our work provides a direct embedding of the EVM in Isabelle and thus might be easier to understand and reason about. Second, our work formalizes cross-contract execution and covers all current EVM opcodes, providing a more complete semantics of the EVM.

6 Conclusion

We introduced Isabelle/EVM, a new formalization of the Ethereum Virtual Machine within Isabelle/HOL. Our formalization is executable, supports cross-contract execution, and implements the complete set of EVM opcodes along with two concrete gas schedules – Shanghai and Cancun. The design is extensible, making it straightforward to incorporate

future hard forks. In addition, the formalization integrates with an optimized code generator, enabling the export of verified Isabelle programs to corresponding Haskell code. Using this mechanism, we validated our model against the official Ethereum test suite to ensure conformance with the specification.

The formalization takes the form of a deep embedding of the EVM: EVM bytecode is modeled explicitly as a datatype in Isabelle rather than as native Isabelle functions. Although we initially considered a shallow embedding, the dynamic behavior of `JUMP` instructions makes them difficult to represent cleanly using a state-monad-based approach. A deep embedding, by contrast, enables reasoning about the EVM language as a whole, not just about individual programs. The trade-off is that proofs about specific programs can be more involved than in a shallow embedding.

There remain several limitations that we plan to address in future work. Currently, the formalization supports only two hard forks. While this suffices to illustrate how additional forks can be integrated, extending support to newer network upgrades – such as the Prague hard fork[7] – would be an important next step. Furthermore, as noted in Section 4, we do not yet model precompiled contracts; incorporating them presents another promising direction for development. Verified compilation from Isabelle/Solidity is an additional area of interest: refinement techniques could ensure that properties proven for Solidity programs carry over to the generated EVM bytecode. Our testing framework can also be refined. A natural improvement would follow the approach used in the Dafny EVM formalization [8], where test cases derived from Ethereum consensus tests are executed using Geth's `evm` tool[8] to produce full execution traces – including intermediate EVM states after each step – rather than just final states.

This formalization opens the door to several applications. Most immediately, it supports bytecode-level verification of smart contracts, enabling proofs about the exact programs executed on the EVM rather than high-level abstractions. It also provides a basis for verifying bytecode optimizers, which aim to transform EVM code into semantically equivalent but more gas-efficient versions. Similar work already exists in Coq [4], and Isabelle/EVM could underpin related efforts in Isabelle. Finally, a formalization of Yul in Isabelle would be a natural extension; the existing Lean formalization [20] could serve as a useful reference point.

References

1 Hayden Adams, Moody Salem, Noah Zinsmeister, Sara Reynolds, Austin Adams, Will Pote, Mark Toda, Alice Henshaw, Emily Williams, and Dan Robinson. Uniswap v4 Core [Draft], June 2023. Accessed: 2025-12-04. URL: `https://raw.githubusercontent.com/jtriley-eth/v4-core/main/whitepaper-v4-draft.pdf`.

2 Asad Ahmed and Diego Marmsoler. Isabelle/Solidity: A tool for the verification of solidity smart contracts (tool paper). In Diego Marmsoler and Meng Xu, editors, *6th International Workshop on Formal Methods for Blockchains, FMBC 2025, Hamilton, Canada, May 4, 2025*, volume 129 of *OASIcs*, pages 12:1–12:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. `doi:10.4230/OASIcs.FMBC.2025.12`.

3 Wolfgang Ahrendt and Richard Bubel. Functional Verification of Smart Contracts via Strong Data Integrity. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 9–24, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-61467-6_2`.

[7] `https://blog.ethereum.org/2025/04/23/pectra-mainnet`
[8] `https://geth.ethereum.org/`

4 Elvira Albert, Samir Genaim, Daniel Kirchner, and Enrique Martin-Martin. Secure Optimizations on Ethereum Bytecode Jump-Free Sequences. *IEEE Transactions on Dependable and Secure Computing*, 22(4):3676–3691, 2025. doi:10.1109/TDSC.2025.3536803.

5 Sidra Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, pages 66–77, Los Angeles, CA, USA, 2018. ACM. doi:10.1145/3167084.

6 Vitalik Buterin and Martin Swende. EIP-2930: Optional access lists. https://eips.ethereum.org/EIPS/eip-2930, 2020. Accessed: 2025-11-06.

7 Franck Cassez. ByteSpector: A Verifying Disassembler for EVM Bytecode. In Diego Marmsoler and Meng Xu, editors, *6th International Workshop on Formal Methods for Blockchains (FMBC 2025)*, volume 129 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:15, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2025.4.

8 Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio M. A. Quiles. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*, pages 571–583, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-27481-7_32.

9 Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. Deductive Verification of Smart Contracts with Dafny. In *Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings*, pages 50–66, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-15008-1_5.

10 CertiK. Vyper Incident Analysis, 2023. Accessed 2025-12-01. URL: https://www.certik.com/resources/blog/vyper-incident-anaylsis.

11 Certora. Certora Prover White Paper. https://www.certora.com/blog/white-paper, 2025.

12 Ethereum Foundation. ethereum/tests: Common tests for all Ethereum implementations. https://github.com/ethereum/tests. Accessed: 2025-11-06.

13 Emilio Frangella and Lasse Herskind. Aave V3 Technical Paper, January 2022. Accessed: 2025-12-04. URL: https://raw.githubusercontent.com/aave-dao/aave-v3-origin/main/docs/Aave_V3_Technical_Paper.pdf.

14 Tomer Ganor, Lior Oppenheim, Gad Elbaz, Pamina Georgiou, and Alex Van Der Lelie. Breaking Down the Balancer Hack, November 2025. Accessed 2025-12-04. URL: https://www.certora.com/blog/breaking-down-the-balancer-hack.

15 Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories. *https://isabelle.in.tum.de/doc/codegen.pdf*, 2026.

16 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi:10.1109/CSF.2018.00022.

17 Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *WTSC'17: 1st Workshop on Trusted Smart Contracts, International Conference on Financial Cryptography and Data Security*. Springer, 2017. doi:10.1007/978-3-319-70278-0_33.

18 Diego Marmsoler, Asad Ahmed, and Achim D. Brucker. Secure smart contracts with Isabelle/Solidity. In Alexandre Madeira and Alexander Knapp, editors, *Software Engineering and Formal Methods - 22nd International Conference, SEFM 2024, Aveiro, Portugal, November 6-8, 2024, Proceedings*, volume 15280 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2024. doi:10.1007/978-3-031-77382-2_10.

19 Diego Marmsoler and Achim D. Brucker. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. *Formal Aspects Computing*, October 2024. doi:10.1145/3700601.

20 Nethermind. EVMYulLean: Executable formal model of the EVM and Yul in Lean 4, 2025. Accessed: 2025-12-04. URL: https://github.com/NethermindEth/EVMYulLean.

21 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.

22 Grigore Rosu. *K: A semantic framework for programming languages and formal analysis tools*, pages 186–206. IOS Press, Netherlands, 2017. Publisher Copyright: © 2017 The authors and IOS Press. All rights reserved. doi:10.3233/978-1-61499-810-5-186.

23 Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical report, Ethereum Foundation, 2014. Version 1.0. URL: https://ethereum.github.io/yellowpaper/paper.pdf.