Verity: A Formally Verified Smart Contract Compiler in Lean 4

Anonymous

Abstract

Smart contracts manage billions of dollars in digital assets, yet most are deployed without formal compiler-correctness guarantees. Existing verification tools usually reason either at the source level or at the bytecode level, leaving a semantic gap between high-level specifications and deployed artifacts.

We present Verity, a Lean 4 Embedded Domain-Specific Language (EDSL) for writing Ethereum Virtual Machine (EVM) smart contracts with a mechanically verified compilation chain to Yul. Contracts are written in monadic Lean 4; the compiler lowers them through a specification model and intermediate representation to Yul, with each stage's correctness proved in Lean's type theory. The verified boundary stops at Yul: the Solidity compiler, the EVM execution model, and gas are part of an explicit trusted computing base with 1 project-specific axiom modeling Keccak-256 function selectors.

We describe Verity's EDSL design, compilation semantics, correctness theorems, explicit trust boundary, and evaluation on representative contracts including a ledger and a token with access control.

1 Introduction

Smart contracts on the Ethereum blockchain collectively manage over one hundred billion dollars in digital assets. Unlike traditional software, deployed contracts are immutable: once on-chain, a bug cannot be patched. The consequences are severe; the 2016 DAO exploit drained $60M from a single contract due to a reentrancy vulnerability [7], and surveys document dozens of exploitable patterns in production contracts [2].

Existing approaches to smart contract assurance fall into two broad categories. *Post-hoc analysis tools* such as Slither [9] and Mythril [14] perform static analysis or symbolic execution on already-compiled contracts. These scale well but provide no formal correctness guarantees and can miss bugs beyond their analysis depth. *Formal verification frameworks* such as KEVM [10] and the Certora Prover [6] verify contracts against specifications, but operate at a single level of abstraction (bytecode or source, respectively) and do not verify the compilation process itself.

A third approach, *verified compilation*, eliminates compiler-introduced bugs within the verified fragment by proving that the compiler preserves program semantics. CompCert demonstrated this for C [12], CakeML for ML [11], and recently Avigad et al. showed that proof-producing compilation is feasible for blockchain applications [3]. To our knowledge, no previous system provides a Lean-based EDSL with a mechanically verified compilation chain targeting Yul for EVM deployment.

We present Verity, a formally verified smart contract compiler implemented and verified in Lean 4 [8]. Verity compiles contracts written in an embedded domain-specific language (EDSL) through three verified layers to Yul, a typed intermediate language with structured control flow designed as a portable compilation target for EVM-targeting languages [17]. The Solidity compiler (`solc`) then compiles Yul to EVM bytecode; this last step, along with gas (the metering mechanism that bounds EVM computation) and EVM runtime semantics, forms an explicit trusted computing base. Our contributions are:

1. A Lean 4 EDSL for EVM smart contracts with a mechanically verified compilation chain through specification, intermediate representation (IR), and Yul code generation, with each stage's correctness proved in Lean's type theory.
2. An explicit trust boundary that delineates what is proved (EDSL through Yul) from what is assumed (the Lean kernel, `solc`, EVM semantics, and gas), with a single project-specific axiom modeling Keccak-256 function selectors.
3. An evaluation on representative contracts (ledger, token with access control) demonstrating end-to-end verified compilation and contract-specification proofs.

The goal is to cover the full EDSL; the current snapshot proves a concrete supported fragment (Section 3), and the fragment expands as additional statement forms are incorporated into the generic proof. Contract-specification proofs (e.g., "only the owner can mint") are a separate concern and remain naturally contract-specific.

The central result can be stated informally as follows.

> **Main theorem (informal).** For every contract C in Verity's supported EDSL fragment, and for every initial state and transaction, the Yul program produced by Verity's compiler has the same observable behavior (success/revert status, returned data, and final storage state) as the source EDSL semantics of C, modulo the trusted assumptions listed in Section 4.3. Event-log equality and gas costs are not compared.

The strongest generic mechanized theorems concern the compiler core from `CompilationModel` through IR to Yul. The source-level theorem applies to a concrete supported fragment rather than to all EDSL programs, and the transfer from Yul to deployed bytecode remains conditional on trusted assumptions about `solc` and the target EVM. Section 2.2 delineates what is in scope and what is not.

Section 2 introduces Verity's architecture with a running example. Section 3 details the compilation pipeline and supported fragment. Section 4 presents the correctness theorems, axioms, and trust boundary. Section 5 discusses related work, and Section 6 covers limitations and future work.

2 Verity's Architecture

This section introduces Verity's compilation pipeline through a running example, a simple counter contract, and gives a bird's-eye view of the architecture before the detailed treatment in Sections 3–4.

2.1 Pipeline

Verity's compilation pipeline transforms contracts through four stages. The first three stages are formally verified within Lean's type theory; the fourth is delegated to `solc`.

EDSL. Contracts are written in a Lean 4 embedded DSL using monadic `do`-notation. The `Contract` type is a state monad over an abstract EVM state (storage, sender, events).

CompilationModel. The compiler-facing contract model generated from the EDSL. It captures public functions, their storage effects, and the observational correspondence used at the source/model boundary.

IR. A lowered intermediate representation with flat control flow and explicit EVM storage/memory operations.

Yul. The final output is Yul source code, which `solc` compiles to EVM bytecode. This last step is outside the verified boundary.

```
-- Storage layout
def count : StorageSlot Uint256 := ⟨0⟩

-- Increment the counter by 1
def increment : Contract Unit := do
  let current ← getStorage count
  setStorage count (add current 1)

-- Decrement the counter by 1
def decrement : Contract Unit := do
  let current ← getStorage count
  setStorage count (sub current 1)

-- Read the current count
def getCount : Contract Uint256 := do
  getStorage count
```

Listing 1: Counter contract in Verity's EDSL. The `Contract` monad provides `getStorage` and `setStorage` for EVM storage access.

2.2 Scope

Verity's correctness guarantees apply to contracts expressible in the supported source fragment (Table 1). The following are explicitly *outside* the current scope:

- **Full Solidity coverage.** Inline assembly, `CREATE2`, and arbitrary post-Yul backend rewrites are not supported.
- **Gas reasoning.** The formal model does not account for EVM gas consumption; semantic correctness does not imply gas-bounded liveness.
- **Yul-to-bytecode verification.** The compilation from Yul to EVM bytecode by `solc` is trusted but unverified.
- **Event-log equality.** The cross-layer comparison does not compare event logs.

The supported fragment is actively expanding; features that compile and execute correctly but are not yet covered by the generic proof are listed in Table 1.

2.3 Running Example: A Counter Contract

We illustrate the pipeline with a counter contract that supports `increment`, `decrement`, and `getCount` operations. Listing 1 shows the EDSL definition.

The contract reads like idiomatic Lean code. The `Contract` monad encapsulates EVM state (storage slots, the transaction sender, and an event log) so that contract logic remains purely functional. Arithmetic is over `Uint256` (unsigned 256-bit integers with wrapping modular semantics, matching EVM behavior).

Section 3 shows how each layer transforms this contract, and Section 4 states the theorem guaranteeing that the compiled Yul faithfully implements the EDSL semantics. Separate contract-specific proofs establish that the EDSL implementation satisfies the human-written specification for the contract.

3 The Verity Compiler

This section describes each compilation layer, the intermediate representations, and the key design decisions. Correctness theorems and the verification methodology are deferred to Section 4.

3.1 Layer 1: EDSL to CompilationModel

The EDSL provides a monadic interface for writing smart contracts in Lean 4. The central type is:

```
abbrev Contract α := ContractState → ContractResult α
```

where `ContractState` models EVM storage, the transaction sender, message value, block timestamp, and an append-only event log.

Layer 1 extracts a `CompilationModel` from EDSL definitions. This compiler-facing model enumerates the contract's public functions, their effect on storage, and the correspondence between EDSL computations and model-level interpretations. This extraction is the frontend semantic bridge: it proves that executing the generated EDSL contract and interpreting the generated compiler-facing model agree on observables.

The separate proofs under `Contracts/<Name>/Proofs/` are contract-specification proofs (see Section 4), not compiler-layer proofs.

3.2 Layer 2: CompilationModel to IR

The specification is lowered to an intermediate representation (IR) with flat control flow and explicit EVM operations. The IR makes storage reads, writes, and control flow explicit while retaining enough structure for compositional reasoning.

The compilation function has type:

```
def compile (spec : CompilationModel)
    (selectors : List Nat)
    : Except String IRContract
```

Selectors are ABI function selectors (4-byte hashes of function signatures) used for runtime dispatch. Their computation relies on the selector axiom within the current project-axiom set (Section 4.2).

Internally, `compile` is decomposed into `validateCompileInputs` (input validation) and `compileValidatedCore` (the actual lowering), exposing the validated intermediate state for proof access.

The generic Layer 2 theorem consumes a `SupportedSpec` witness that restricts the model to the currently proved fragment: no custom errors, no constructor, and normalized storage fields. Each function must satisfy `SupportedFunction` (supported parameter types and body). The compiler itself may still compile contracts outside this fragment, but those broader cases are not yet covered by the generic theorem and are instead surfaced through diagnostics and trust-boundary reports (e.g., `--deny-unchecked-dependencies`). The goal is to extend the proved fragment to cover the full EDSL; the current fragment (Table 1) expands as additional statement forms are proved.

3.3 Layer 3: IR to Yul

The final verified stage emits Yul. The code generator traverses the IR and produces a `YulObject` containing deploy and runtime code:

```
def emitYul (contract : IRContract) : YulObject :=
  { name := contract.name,
    deployCode := deployCode contract,
    runtimeCode := runtimeCode contract }
```

Table 1: EDSL feature support. "Operational" means the feature is supported in Spec/Codegen/Interpreter. "Generic proof" indicates coverage by the `SupportedSpec` Layer 2 theorem.

Feature	Operational	Generic proof
Core arithmetic, booleans, local variables	supported	proved
Require/revert, return, stop	supported	proved
Storage read/write (uint256 fields)	supported	proved
Selector dispatch + ABI loading	supported	proved
Payable modifier, event ABI parity	supported	proved
If/else branching	supported	partial
Storage (address fields, mappings, packed, arrays, structs)	supported	not in generic proof
Internal helper calls	supported	not compositional
forEach loops	supported	known proof gap
Constructor, fallback/receive	supported	excluded
Events (emit), raw log	supported	not in proof model
Custom errors, multi-return, returnValues	supported	not in generic proof
Linked externals, ECM	supported	not in generic proof
Low-level calls, delegatecall, returndata	supported	not in proof model
Linear memory (mstore, mload, calldatacopy)	supported	partially modeled
Bitwise ops, signed arithmetic, fixed-point math	supported	not in generic proof
keccak256	supported	axiomatized
ABI JSON artifact generation	supported	n/a
ETH balance tracking, create/create2, mixins	not yet implemented	

Table 2: Verification layers in the Verity compilation pipeline.

Layer	**Stage**	**Status**	**Axioms**
1	EDSL -> CompilationModel	Fully verified	none
2	CompilationModel -> IR	Generic (supported fragment)	none
3	IR -> Yul	Generic (axiom-dependent)	`keccak256_first_4_bytes`
ext	Yul -> bytecode (solc)	Trusted (0.8.33+commit.64118f21)	N/A (unverified)

The generated Yul is subsequently compiled to EVM bytecode by `solc` (v0.8.33), which lies outside the verified boundary.

3.4 Supported Source Fragment

Table 1 summarizes each EDSL feature and distinguishes *operational* support (the feature compiles and executes correctly) from *generic proof* coverage (the feature is covered by the `SupportedSpec` whole-contract theorem in Layer 2). Most features work end-to-end; the generic proof covers a narrower fragment that is actively expanding.

Formally, the proved fragment is characterized by the `SupportedSpec` predicate used in the Layer 2 theorem; the key distinction is between features that compile and execute correctly and features covered by the generic proof.

Table 2 summarizes the verification status of each layer. Layers 1 and 2 carry no project-specific axioms. Layer 3 depends on the selector axiom (Section 4.2). The final stage trusts `solc`.

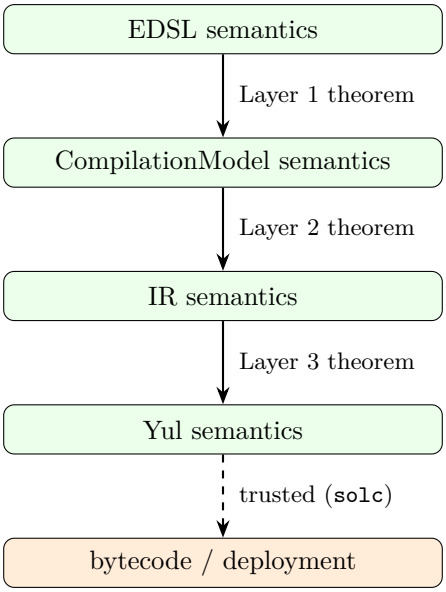

Figure 1: Proof structure of Verity's verified compilation chain. Semantic equivalence is mechanically verified (solid arrows) from the EDSL through Yul; the dashed arrow marks the trusted backend boundary.

4 Formal Verification

This section presents the correctness theorems, the project-axiom set, and the trust boundary.

4.1 Correctness Theorems

The central correctness property states that compiling through the three proved layers and interpreting the result produces the same observable behavior as executing the source contract directly. Figure 1 shows where semantic equivalence is established and where trust begins.

The argument has two levels of generality: the middle and backend compiler theorems are generic, while the source-level theorem applies to the supported source fragment identified in Table 1.

We distinguish two kinds of proof in Verity. *Compiler-layer proofs* (Layers 1–3) establish that compilation preserves semantics for all contracts in the supported fragment. *Contract-specification proofs* (under `Contracts/<Name>/Proofs/`) establish that a particular EDSL implementation satisfies its human-written specification (e.g., "only the owner can mint"). These are independent: compiler proofs apply to every supported contract, while specification proofs are contract-specific.

Cross-layer comparison uses the `resultsMatch` relation: two execution results are related if they agree on success/revert status, returned data, and post-rollback storage state. Event-log equality and gas costs are not compared.

Table 3 makes explicit what kind of result Verity provides. The strongest generic mechanized result is the compiler core from `CompilationModel` to Yul. The source-level result is intentionally stated over Verity's supported fragment rather than over all EDSL programs.

We illustrate with the counter contract from Section 2.3. Listing 2 shows the Spec-level correctness theorem for the `increment` function.

The theorem quantifies universally over all contract states and senders. The proof proceeds by unfolding definitions, simplification, and reflexivity (`rfl`), since the EDSL execution and spec interpretation reduce to definitionally equal terms. This pattern is representative of Verity's contract-specific specification proofs: they show that a given contract implementation satisfies

Table 3: Status and generality of Verity's main correctness claims.

Claim layer	Generality	What is mechanized
Source → `CompilationModel`	Source-fragment theorem schema	Frontend lowering for the supported source fragment and its semantic-preservation obligations
`CompilationModel` → IR	Generic compiler theorem	Lowering correctness for arbitrary well-formed compilation models
IR → Yul	Generic compiler theorem	Yul code-generation correctness for arbitrary well-formed IR contracts
Yul → bytecode	Conditional transfer only	Not mechanized; relies on trusted backend and deployment assumptions

its stated behavior, rather than restating the generic compiler layer theorem.

To make the verified compilation chain explicit, Listing 3 shows the Layer 3 preservation theorem used to connect IR semantics to emitted Yul semantics for any transaction and initial state.

Beyond the selector bound, the theorem now requires three structural preconditions: `hNoWrap` ensures the ABI-encoded calldata size $(4 + |args| \cdot 32)$ does not wrap modulo 2^{256}, `hWF/hNoFallback/hNoReceive` assert well-formedness of the contract structure for the supported fragment, and `hdispatchGuardSafe` ensures per-function dispatch guard safety for the given transaction.

The `hbody` hypothesis is addressed by `ir_function_body_equiv` (same file), which derives per-function IR/Yul agreement from statement-level equivalence and a fuel-adequacy lemma, yielding a self-contained result for any function, argument list, and initial state. The two theorems use different Yul entry points (`interpretYulBody` vs. `interpretYulBodyFromState`); bridging the entry-point gap completes the discharge.

At the contract level, the end-to-end argument composes (i) EDSL↔`CompilationModel` correctness lemmas (Layer 1), (ii) `CompilationModel`→IR equivalence (Layer 2), and (iii) the IR→Yul preservation theorem above (Layer 3). The trusted boundary begins at `solc` for Yul→bytecode.

4.2 Project Axioms

At the pinned snapshot, Verity uses 1 project-specific axiom beyond Lean's built-in foundational axioms (`propext`, `Quot.sound`, `Classical.choice`). All compiler-layer correctness facts (Layers 1–3) are proved theorems; the sole remaining axiom is:

```
axiom keccak256_first_4_bytes
    (sig : String) : Nat
```

This axiom models the computation of EVM function selectors (`bytes4`(`keccak256`(*signature*))), used in ABI dispatch tables. Keccak-256 is a cryptographic hash function whose implementation would require substantial effort to formalize in Lean's logic. We treat Keccak-256 as an external oracle and validate it operationally. CI cross-checks every computed selector against `solc -hashes`, and fixture regression tests detect any drift.

4.3 Trust Boundary

Rather than claiming end-to-end correctness from source to deployed bytecode, Verity enumerates precisely what is proved and what is assumed.

```
theorem counter_increment_correct
    (state : ContractState)
    (sender : Address) :
    let edslFinal :=
      (increment.runState
        { state with sender := sender })
    let specTx : Transaction := {
      sender := sender,
      functionName := "increment",
      args := [] }
    let specResult :=
      interpretSpec counterSpec
        (counterEdslToSpecStorage state)
        specTx
    specResult.success = true ∧
    specResult.finalStorage.getSlot 0 =
      (edslFinal.storage 0).val := by
  unfold increment counterSpec
    interpretSpec
    counterEdslToSpecStorage
    Contract.runState
  simp [add, count, execFunction,
    execStmts, execStmt, evalExpr,
    SpecStorage.setSlot,
    SpecStorage.getSlot, modulus]
  rfl
```

Listing 2: Correctness theorem for the counter's `increment` function. The theorem states that the EDSL execution and the spec-level interpretation agree on the final storage.

Table 4: Project axioms beyond Lean's built-in axioms.

Axiom	Location	Mitigations
`keccak256_first_4_bytes`	`Compiler/Selectors.lean:41`	CI cross-check against solc –hashes; Selector fixture regression tests; Compilation/tests fail on selector drift

What is proved. The EDSL-to-Yul compilation chain (Layers 1–3) is mechanically verified within Lean's type theory. Every contract in the supported fragment inherits the compilation-correctness guarantee without per-contract proof effort. The proof depends on a single project axiom (the selector oracle, Section 4.2).

What is assumed (TCB). The trusted computing base consists of components whose correctness is assumed:

Lean kernel. The type-checker that validates all proofs. This is the standard assumption for any Lean-based verification.

`solc` (v0.8.33). The Solidity compiler, which compiles Yul to EVM bytecode. Version-pinned in `foundry.toml` and enforced by CI.

EVM semantics. The execution model of the Ethereum Virtual Machine. Gas consumption is not modeled; semantic correctness does not imply gas-bounded liveness.

Linked Yul libraries. External functions injected by the linker are outside formal proofs and must be audited separately.

```
theorem yulCodegen_preserves_semantics
    (contract : IRContract)
    (tx : IRTransaction)
    (initialState : IRState)
    (hselector : tx.functionSelector < selectorModulus)
    (hNoWrap : 4 + tx.args.length * 32 < evmModulus)
    (hWF : ContractWF contract)
    (hNoFallback : contract.fallbackEntrypoint = none)
    (hNoReceive : contract.receiveEntrypoint = none)
    (hdispatchGuardSafe : forall fn,
      fn in contract.functions ->
        DispatchGuardsSafe fn tx)
    (hbody : forall fn, fn in contract.functions ->
      resultsMatch
        (execIRFunction fn tx.args
          { initialState with sender := tx.sender,
            calldata := tx.args,
            selector := tx.functionSelector })
        (interpretYulBody fn tx
          { initialState with sender := tx.sender,
            calldata := tx.args,
            selector := tx.functionSelector })) :
    resultsMatch
      (interpretIR contract tx initialState)
      (interpretYulFromIR contract tx initialState) := by
  ...
```

Listing 3: Layer 3 preservation theorem (`IR -> Yul`).

Verifying `solc` would require formalizing the full Yul-to-bytecode compilation, which is a separate research challenge. Instead, Verity targets Yul as the verified boundary because it is close to the deployed artifact while still structured enough to admit reusable compiler proofs.

Semantic caveats. Two modeling choices affect the interpretation of results: (1) `Uint256` arithmetic in the formal model is wrapping modulo 2^{256}, matching EVM behavior but differing from Solidity's default checked arithmetic; and (2) the formal model may expose intermediate state in reverted computations, whereas the EVM discards state on revert. These caveats should be read together with Table 3: the paper proves semantic preservation for the modeled observables of the proof-backed path, not full deploy-time fidelity for every EVM concern.

4.4 Evaluation

The evaluation addresses four questions: (1) What kinds of contracts can Verity handle? (2) What parts of the compiler proof are generic vs. contract-specific? (3) What effort is required to verify a new contract? (4) What is the role of testing alongside formal proofs?

Verified contracts. Verity has been applied to 11 contract families spanning counters, ERC-20 and ERC-721 tokens, ownership, reentrancy guards, ledger conservation, and simple storage. Two of these (Ledger and SimpleToken) serve as detailed case studies below.

Scope of the supported fragment. Table 1 (Section 3) lists the EDSL features that compile and execute correctly and those covered by the generic compiler proof. The supported fragment spans storage-heavy contracts with mappings, tuple-valued interfaces, internal helpers, events,

Table 5: Trusted computing base: components outside formal verification.

Component	Role	Trust Type
Solidity Compiler (solc)	Compiles Yul → EVM bytecode.	external tool
Selector Oracle (keccak256)	Single project axiom: models function-selector computation (bytes4(keccak256(signature))); validated operationally by CI cross-check against solc --hashes.	axiom
Linked Yul Libraries	External functions injected at compile time (e.g., Poseidon hash).	operational
Mapping Slot Derivation	keccak256(abi.encode(key, baseSlot)) for Solidity-compatible storage (activeMappingSlotBackend = .keccak).	operational
EVM/Yul Semantics and Gas	Runtime execution model.	operational
External Call Modules (ECMs)	Reusable typed external call patterns (ERC-20 writes/ reads including totalSupply, ERC-4626 preview/ conversion helpers plus totalAssets, asset, max* limit reads, and deposit, oracle reads, precompiles, callbacks).	operational
Lean Kernel	Proof checker soundness. Foundational assumption for all Lean-based verification.	foundational
Macro Elaborator (verity_contract)	Generates both EDSL Contract monad value and CompilationModel from one syntax tree.	operational
Local Unsafe / Refinement Obligations	Let a function or constructor declare a localized proof obligation for an unsafe/assembly-shaped boundary without marking the whole contract as opaque.	operational

and access control. Features outside the generic proof (loops, custom errors, linked externals) compile and are tested but require per-contract proof obligations.

Generic vs. contract-specific proofs. Compiler-layer proofs (Layers 1–3) are proved once and apply to every contract in the supported fragment. Contract-specification proofs (e.g., "only the owner can mint") are necessarily contract-specific but reuse shared proof infrastructure (storage-slot reasoning, arithmetic lemmas, revert-preservation). Adding a new contract to the verified corpus requires writing the EDSL definition and its specification proofs; the compiler correctness is inherited. As a concrete measure, the Ledger and SimpleToken case studies required 33 and 61 contract-specific theorems respectively, on top of the shared compiler infrastructure.

Proof completeness. At the pinned snapshot, the codebase contains zero instances of `sorry` or `admit`, confirmed by automated scanning of all 239 Lean source files.

Differential testing. Beyond formal verification, the project maintains 520 Foundry integration tests across 49 suites, providing differential testing between the EDSL interpreter and `solc`-compiled EVM execution. These tests serve as an independent cross-check of the formal model against the deployed artifact.

Ledger case study. At the source/spec boundary, Verity proves that successful `deposit`, `withdraw`, `transfer`, and read-only `getBalance` executions agree with the specification on success and on the relevant observable storage cells. It also proves that insufficient-balance `withdraw` and `transfer` calls preserve revert behavior. Beyond per-function agreement, the library includes exact sender/recipient balance-conservation lemmas for successful transfers and isolation lemmas showing that unrelated accounts are unchanged by `deposit`. This makes `Ledger` a compact example where the verified path simultaneously covers functional correctness, local non-interference, and quantitative state reasoning.

Table 6: Contract-level guarantees highlighted by the two evaluation case studies.

Contract	Source-level guarantees highlighted in the paper	Non-trivial aspect	Scope condition
`Ledger`	Successful `deposit`, `withdraw`, `transfer`, and `getBalance` agree with the specification; insufficient-balance executions preserve revert behavior; transfers preserve the sender/recipient balance sum; deposits leave unrelated accounts unchanged.	Combines state updates, reverts, local isolation, and quantitative conservation.	Assumes sufficient sender balance and no recipient overflow for successful transfer theorems.
`SimpleToken`	Owner-authorized `mint` agrees with the specification on beneficiary balance and `totalSupply`; non-owner `mint` reverts; successful `transfer` agrees on sender/recipient balances; transfers preserve `totalSupply`; successful `mint` implies the caller is the owner.	Combines access control, arithmetic side conditions, and coupled balance/supply reasoning.	Exact per-pair balance equations; no global sum invariant.

SimpleToken case study. `SimpleToken` adds access control and supply accounting to mapping-backed balances. Verity proves that owner-authorized `mint` updates both the beneficiary balance and `totalSupply` as specified, that non-owner `mint` attempts revert, that successful `transfer` preserves agreement on sender and recipient balances, and that transfers preserve `totalSupply`. The development also proves an explicit access-control property: if `mint` succeeds, then the caller was the recorded owner. For balance conservation, the theorems state exact balance equations rather than the superficially appealing invariant "the sum of all balances equals total supply," which is false when the address list contains duplicates. Verity instead states exact per-pair balance equations that remain valid regardless of address-list structure.

Reproducibility. All claims are pinned to a specific Verity commit; `make regen && make check` regenerates every table and fails closed if any claim lacks supporting artifacts.

5 Related Work

Verified compilers. The verified compilation paradigm was established by CompCert [12], which provides a mechanically verified C compiler in Coq, and CakeML [11], a verified ML implementation in HOL4. Appel's Verified Software Toolchain [1] extends this approach to verified linking and composition. Verity follows the same paradigm, proving the compiler correct once so that every compiled program inherits the guarantee, but targets a different domain (smart contracts) and a different proof assistant (Lean 4).

Proof-producing compilation for blockchains. The closest related work is by Avigad et al. [3, 4], who extended the CairoZero compiler with tooling that generates, for each compiled program, a Lean 3 proof that the machine code meets a high-level specification. Their approach verifies *specific compiled programs* (ECDSA cryptographic primitives, dictionary operations) after compilation, requiring per-program proof effort. Verity takes a complementary approach: we verify *the compiler itself*, so that every contract expressible in the EDSL inherits correctness without additional proof work. The tradeoff is that Verity's guarantees are limited to contracts within the EDSL's expressiveness, whereas Avigad et al.'s approach can in principle verify any compiled Cairo program.

EVM formal semantics. Hildenbrandt et al. [10] formalize a complete executable semantics of the EVM within the K framework [15], enabling verification of EVM bytecode via reachability logic. KEVM operates at the bytecode level, requiring specifications to be written against low-level EVM state. Verity operates at a higher abstraction level: users write specifications as Lean propositions against the EDSL, and the verified compiler bridges the gap to bytecode. Verity's choice of Yul as a verified target reflects a practical tradeoff: Yul is close to deployable artifacts, but still structured enough to support reusable compiler proofs.

Smart contract verification tools. DeepSEA [16] is a verified smart contract language targeting the EVM, implemented and verified in Coq. Its compiler is proved correct in Coq, and it additionally generates a formal Coq model alongside the bytecode against which users can prove functional properties of specific contracts. The key differences from Verity are: (i) proof assistant (Coq vs. Lean 4), (ii) target representation (EVM bytecode vs. Yul), and (iii) workflow, since Verity embeds the contract language directly in Lean 4 as an EDSL, so that contract specifications, compiler proofs, and contract-specific proofs all live in a single language and proof environment.

The Certora Prover [6] verifies Solidity contracts against CVL (Certora Verification Language) specifications using SMT-based bounded model checking. Slither [9] performs static analysis on Solidity source, and Mythril [14] uses symbolic execution to find common vulnerability patterns. These tools automate common checks but offer weaker guarantees than mechanized proofs: they may miss bugs beyond their analysis depth or loop-unrolling bounds. The key distinction is where proof obligations are paid: Verity invests in reusable compiler proofs, whereas many contract-verification tools pay more of the cost at the individual contract, query, or symbolic-execution boundary.

Other blockchain formal methods. Bernardo et al. [5] develop Mi-Cho-Coq, a framework for certifying Tezos smart contracts in Coq, operating at the Michelson bytecode level. This is analogous to KEVM's approach but for a different blockchain. Verity differs in targeting the compilation process rather than post-hoc bytecode verification.

Lean 4 as a verification platform. Lean 4 [8] is both a programming language and a proof assistant, which is well suited for verified compiler construction since the compiler and its proofs coexist in a single language. The Mathlib library [13] provides a growing mathematical foundation. Verity suggests that Lean 4 is a viable platform for mechanically verified smart-contract compilation.

Summary of Verity's positioning. Verity combines several design choices:

- *Lean-based EDSL*, not an external source language;
- *mechanically verified compiler chain*, not only per-program proofs;
- *target boundary at Yul* for EVM deployment, not bytecode;
- *explicit trust boundary* with enumerated axioms;
- *source-level contract proofs* living in the same proof assistant ecosystem as the compiler proofs.

6 Limitations and Future Work

Scope of the EDSL. The supported source fragment is described in Section 2.2. Extending the proved fragment toward full EDSL coverage is ongoing work.

The solc trust boundary. The compilation from Yul to EVM bytecode relies on solc, which is trusted but unverified. A bug in solc could produce bytecode that does not faithfully implement the verified Yul. Mitigations include version pinning and differential testing, but the gap remains. Integrating a formal EVM semantics (e.g., building on KEVM [10]) to verify the Yul-to-bytecode step is a natural next step.

Gas modeling. The formal model does not account for EVM gas consumption. Semantic correctness does not imply gas-bounded liveness, since a contract may be correct but run out of gas. Incorporating a formal gas model is future work.

Remaining axiom. At the pinned snapshot, 1 project-specific axiom remains (Table 4): the selector oracle, which models Keccak-256 for ABI dispatch. The axiom is validated operationally by CI; formalizing Keccak-256 in Lean is future work.

Threats to validity. *Construct validity:* theorem and proof counts depend on static classification heuristics and a curated manifest. *Internal validity:* semantic equivalence is proved for the verified EDSL→Yul path, but Yul→bytecode remains trusted via `solc`. *External validity:* results currently cover the modeled EDSL fragment and 11 deployable contract families; claims should not be generalized to unsupported Solidity features or arbitrary EVM programs.

7 Conclusion

Verity is a Lean 4 EDSL for EVM smart contracts with a mechanically verified compilation chain to Yul. The verified boundary covers the EDSL through specification, intermediate representation, and Yul code generation; the trust boundary around `solc`, EVM semantics, and gas is made explicit. Source-level specifications and proofs connect to deployable artifacts through this verified chain, enabling both generic compiler proofs and contract-specific correctness arguments.

The next steps are to extend the supported source fragment toward full EDSL coverage and reduce the trusted boundary around the Yul-to-bytecode backend.

References

[1] Andrew W. Appel. Verified software toolchain. In *Programming Languages and Systems – ESOP 2011*, volume 6602 of *LNCS*, pages 1–17, Berlin, Heidelberg, 2011. Springer.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST 2017)*, volume 10204 of *LNCS*, pages 164–186, Berlin, Heidelberg, 2017. Springer.

[3] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A proof-producing compiler for blockchain applications. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *LIPIcs*, pages 7:1–7:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A proof-producing compiler for blockchain applications. Journal of Automated Reasoning, 2025.

[5] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In *Formal Methods. FM 2019 International Workshops*, volume 12232 of *LNCS*, pages 368–379, Cham, 2019. Springer.

[6] Certora. Certora prover documentation. `https://docs.certora.com/`, 2024.

[7] Phil Daian. Analysis of the DAO exploit. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, 2016.

[8] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635, Cham, 2021. Springer.

[9] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, Montreal, QC, Canada, 2019. IEEE.

[10] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Roşu. KEVM: A complete formal semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, Oxford, UK, 2018. IEEE.

[11] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 179–191, New York, NY, USA, 2014. ACM.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] Mathlib Community. Mathlib4: The Lean 4 mathematical library. `https://github.com/leanprover-community/mathlib4`, 2024.

[14] Bernhard Mueller. Smashing Ethereum smart contracts for fun and actual profit. HITB Security Conference, 2018. Amsterdam, Netherlands.

[15] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[16] Vilhelm Sjöberg, Yuyang Sang, Shu-Chun Weng, and Zhong Shao. DeepSEA: A language for certified system software. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[17] Solidity Team. Solidity documentation. `https://docs.soliditylang.org/`, 2024.