# Verity: A Verified Compiler for a Core Fragment of EVM Smart Contracts in Lean 4

Thomas Marchand[1]

[1]Independent Researcher, Lisbon, Portugal

**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 and *zero* project-specific `axiom` declarations. Contracts are written in monadic Lean 4 using a dedicated `verity_contract` macro. The reusable mechanized compiler proofs cover the `CompilationModel`-to-IR and IR-to-Yul backend for the supported core fragment. The frontend macro emits kernel-checked per-function bridge facts, but the macro elaborator itself remains trusted.

The generic compiler proof covers a *core computational fragment*: storage reads and writes, mappings, arithmetic (including signed and fixed-point), conditional branching, and access control. Features outside this fragment (e.g., events and loops) are operationally supported and regression-tested but require per-contract proof obligations rather than the generic theorem. The verified boundary stops at Yul: the operational trusted computing base includes the Lean kernel, the macro elaborator, Verity's Yul/EVM semantic assumptions, Keccak agreement with the EVM's `keccak256` opcode, `solc` for Yul-to-bytecode, gas, and linked externals or local obligations when used.

We describe Verity's EDSL design, compilation semantics, correctness theorems, explicit trust boundary, and evaluation on representative contracts including proof-complete storage-heavy examples in the supported fragment and operational examples whose additional trust surfaces are explicit.

**Keywords**

formal verification, smart contracts, compiler verification, Lean 4, theorem proving, Ethereum

## 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 reentrancy vulnerability [1], and in July 2023 a compiler bug in Vyper silently broke reentrancy locks, draining approximately $70M from Curve Finance pools despite correct source code [2]. Surveys document dozens of exploitable patterns in production contracts [3].

Existing approaches to smart contract assurance fall into two broad categories. *Post-hoc analysis tools* such as Slither [4] and Mythril [5] 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 [6] and the Certora Prover [7] 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 [8], CakeML for ML [9], and recently Avigad et al. showed that proof-producing compilation is feasible for blockchain applications [10]. To our knowledge, no previous system provides a Lean-based EDSL with a mechanically verified compilation chain targeting Yul for EVM deployment.

*ETHReS: Ethereum Research Symposium @ ETHPrague, May, 8-10 2026 - Prague, Czech Republic*
research@thomas.md (T. Marchand)
0009-0009-2874-4777 (T. Marchand)

We present Verity, a smart contract compiler implemented and verified in Lean 4 [11]. Verity compiles contracts written in an embedded domain-specific language (EDSL) through a macro-generated frontend and two generically verified compiler stages to Yul, an intermediate language with structured control flow designed as a portable compilation target for EVM-targeting languages [12]. The generic compiler proof covers a core computational fragment (storage, arithmetic, branching, mappings, access control); features outside this fragment are operationally supported and regression-tested but require per-contract proof obligations. The Solidity compiler (`solc`) then compiles Yul to EVM bytecode. This last step, the macro elaborator, semantic reference assumptions, Keccak agreement, and gas (the metering mechanism that bounds EVM computation) form an explicit trusted computing base. Verity implements its own Yul interpreter with proven agreement to EVMYulLean [13], an independently maintained Lean 4 formalization validated against the official Ethereum conformance test suite.

Our contributions are:

1. A mechanically verified compiler core with *zero* project-specific `axiom` declarations in Lean, achieved by internalizing all Keccak-256 computations (function selectors, mapping-slot derivations) as kernel-computable theorems via a vendored Keccak engine under `Compiler/Keccak/`. Here "vendored" means that the Keccak implementation is checked-in Lean source, including the generated `CircuitData.lean` unrolled/static Keccak-f[1600] circuit produced from the tracer in `Circuit.lean`, and evaluated by Lean's kernel during proof checking rather than used as an external proof oracle. The remaining trust assumption is extensional: this Lean implementation must agree with the EVM's `keccak256` opcode. CI cross-checks against `solc --hashes` and regression suites are testing evidence for that agreement, not a mechanized proof of the Keccak permutation. An explicit trust boundary delineates what is proved from what is assumed; the operational trusted computing base includes the Lean kernel, the `verity_contract` macro elaborator, `solc`, EVM/Yul semantic assumptions including EVMYulLean [13], gas, and linked externals or local obligations when used (Section 6.3).
2. A Lean 4 EDSL for EVM smart contracts with a three-layer compiler through a deep-embedding `CompilationModel`, an intermediate representation (IR), and Yul code generation. The `CompilationModel`-to-IR and IR-to-Yul stages are covered by reusable mechanized preservation theorems for a core computational fragment; the frontend macro emits kernel-checked per-function bridge facts while the macro elaborator remains trusted. Contract-level specifications are written as ordinary Lean propositions over the shallow EDSL and discharged by tactic scripts (Section 6.1).
3. An evaluation on representative contracts demonstrating proof-complete storage-heavy examples in the supported fragment and operational examples whose additional trust surfaces are explicit.

We aim to eventually cover the full EDSL; the current snapshot proves a concrete supported fragment (Section 5), 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 $C$ (formalized as the `resultsMatch` relation), modulo the trusted assumptions listed in Section 6.3. Gas costs are excluded from the comparison.

The `resultsMatch` relation also compares emitted events; event-emitting contracts are currently outside the generic supported fragment and require per-contract proof obligations (Section 7).

The strongest generic mechanized theorems concern the compiler core from `CompilationModel` through IR to Yul. The source-level connection is made by bridge facts emitted by the macro and

checked by Lean's kernel for a concrete supported fragment, rather than by a proof of the macro elaborator over all EDSL programs, and the transfer from Yul to deployed bytecode remains conditional on trusted assumptions about `solc` and the target EVM. Section 4.1 delineates what is in scope and what is not.

Section 3 provides necessary background on Lean 4, embeddings, and the EVM. Section 4 introduces Verity's architecture with a running example. Section 5 details the compilation pipeline and supported fragment. Section 6 presents the correctness theorems, axioms, and trust boundary. Section 7 evaluates Verity on representative contracts. Section 2 discusses related work, and Section 8 covers limitations and future work.

## 2. Related Work

**Verified compilers.** The verified compilation paradigm was established by CompCert [8], which provides a mechanically verified C compiler in Coq, and CakeML [9], a verified ML implementation in HOL4. Appel's Verified Software Toolchain [14] 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. [10, 15], 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. [6] formalize a complete executable semantics of the EVM within the K framework [16], enabling verification of EVM bytecode via reachability logic. KEVM operates at the bytecode level, requiring specifications to be written against low-level EVM state. EVMYulLean [13] provides a Lean 4 formalization of EVM and Yul semantics validated against the official Ethereum conformance test suite. Verity implements its own Yul interpreter and proves its pure-builtin semantics equivalent to EVMYulLean's via 15 bridge theorems, giving the compilation proofs an externally auditable semantic reference. Verity operates at a higher abstraction level: users write specifications as Lean propositions against the EDSL, and the verified compiler bridges the gap to Yul.

**Smart contract verification frameworks.** DeepSEA [17] is a certified systems programming language implemented and verified in Coq, with a layered specification approach that uses refinement proofs to connect high-level specifications with low-level implementations. Originally targeting C via CompCert for system software, DeepSEA was subsequently adapted for smart contracts: the later DeepSEA system targets Ethereum and compiles source programs into EVM bytecode and a Coq model for contract proofs [18]. Like Verity, it combines a proof assistant-facing specification artifact with a verified compiler path, but the proof boundary and workflow differ. DeepSEA aims at end-to-end foundational correctness of generated EVM bytecode, with compiler correctness internalized in Coq and gas tracked through the EVM compilation proof. Verity instead verifies a generic Lean EDSL-to-Yul compiler boundary, proves source-level contract properties in the same Lean development as the compiler, and makes the post-Yul path an explicit operational trust assumption through `solc` and deployment infrastructure.

**Table 1**
Comparison of verified smart-contract and compiler systems.

| System | Prover | Source | Target | Axioms | Verification style |
|---|---|---|---|---|---|
| CompCert [8] | Coq | C subset | Assembly | 0 | Verified compiler |
| CakeML [9] | HOL4 | ML | Assembly | 0 | Verified compiler |
| Avigad et al. [10] | Lean 3 | CairoZero | Machine code | 0 | Proof-producing |
| DeepSEA [18] | Coq | DeepSEA | EVM bytecode | 0 | Verified compiler |
| ConCert [19] | Coq | Coq | Liquidity | 0 | Certified extraction |
| KEVM [6] | K | n/a | EVM bytecode | n/a | Post-hoc verification |
| **Verity** | Lean 4 | Lean EDSL | Yul | 0 | Verified compiler |

The difference is therefore not only parser versus EDSL, or bytecode versus Yul. DeepSEA pays for a standalone contract language and compiler stack in exchange for a bytecode-level target theorem. Verity pays for a trusted Lean macro elaborator and a trusted Yul-to-bytecode backend in exchange for tight integration with Lean 4's type checker, tactics, libraries, and editor support, and for reusable compiler theorems over the EDSL representation used directly by contract authors. Verity also does not currently model gas formally, whereas DeepSEA's EVM compiler correctness account includes gas-cost tracking. These choices place different components in the trusted computing base and make the systems complementary rather than simple substitutes. ConCert [19] takes a different approach: contracts are written as native Coq functions and extracted to target languages (Liquidity, CameLIGO) via MetaCoq's certified erasure pipeline. ConCert provides trace-based multi-contract reasoning but leaves the final pretty-printing step to each target language unverified, a gap that Verity's verified compiler avoids within its EDSL-to-Yul boundary.

The Certora Prover [7] verifies Solidity contracts against CVL (Certora Verification Language) specifications via SMT-based verification-condition generation with bounded loop unrolling. Slither [4] performs static analysis on Solidity source, and Mythril [5] 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. [20] 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 [11] 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 [21] 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.** Table 1 summarizes how Verity relates to the most closely comparable systems along key design axes.
Verity combines several design choices that distinguish it from these systems:

- *Lean-based EDSL*, not an external source language;
- *mechanically verified compiler chain*, not only per-program proofs;
- *target boundary at Yul* with pure-builtin equivalence proven against EVMYulLean;
- *explicit trust boundary* (Section 6.3);
- *source-level contract proofs* living in the same proof assistant ecosystem as the compiler proofs.

# 3. Background

**Lean 4.** Lean 4 [11] is both a general-purpose programming language and an interactive theorem prover based on the Calculus of Inductive Constructions. Programs and proofs coexist in a single language: a `theorem` is checked by the same kernel that type-checks ordinary `def` definitions, so any term that type-checks constitutes a valid proof. We use several Lean idioms throughout the paper:

- `structure` defines a record type with named fields; `inductive` defines a sum type with constructors.
- `abbrev` introduces a type alias that the elaborator unfolds transparently.
- Monadic `do`-notation sequences effectful computations, with ← binding intermediate results.
- Proofs proceed by tactic scripts (`simp`, `rfl`, `unfold`, etc.) that discharge goals step by step.

**Shallow vs. deep embeddings.** A *shallow embedding* represents domain constructs as ordinary host-language functions: contract operations are Lean functions, directly executable and available for proof automation. A *deep embedding* represents them as an inductive abstract syntax tree (AST) that a compiler can pattern-match on and transform. Verity generates *both* from a single source definition via the `verity_contract` macro, following the approach of Svenningsson and Axelsson [22], and proves that the two agree on observables.

**The Ethereum Virtual Machine.** The EVM is a stack-based virtual machine that executes smart contracts. Relevant concepts for this paper are: *storage*, a persistent key-value store mapping 256-bit slots to 256-bit values; *calldata*, the immutable transaction input; and *selectors*, the first four bytes of the Keccak-256 hash of a function signature, used for runtime dispatch.

**Yul.** Yul is an intermediate language with structured control flow that serves as a portable compilation target for EVM-targeting languages [12]. Unlike raw EVM bytecode, Yul exposes named local variables, `if`, `for`, `switch`, and function definitions. Its builtins correspond directly to EVM opcodes (`sload`, `sstore`, `add`, `keccak256`, etc.). Verity targets Yul as the verified output because it is close to the deployed artifact while still structured enough to admit reusable compiler proofs.

**The Solidity compiler (`solc`).** `solc` is the reference Solidity compiler maintained by the Ethereum Foundation. In addition to compiling Solidity source, it lowers Yul to EVM bytecode through a series of unverified optimization and encoding passes. Verity pins a specific `solc` version (v0.8.33) in its build configuration and treats this final lowering step as a trusted, unverified backend; the compilation chain that Verity *does* verify terminates at Yul.

# 4. Overview of Verity's Architecture

This section provides a high-level view of Verity's compilation pipeline and verified boundary (summarized in Figure 1), and Section 4.2 walks through a simple counter contract that illustrates the pipeline end-to-end, before the detailed treatment in Sections 5–6.

## 4.1. Pipeline and Scope

Verity's compilation pipeline transforms contracts through the lowering chain `EDSL` → `CompilationModel` → `IR` → `Yul`. The middle and backend stages are covered by generic Lean preservation theorems; the frontend macro emits per-contract bridge facts but remains part of the operational trusted computing base. The fourth stage (Yul to bytecode) 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). This is a *shallow* embedding: contract functions are ordinary Lean functions, directly executable for testing and available for native proof automation.

**CompilationModel.** A *deep* embedding generated from the EDSL by the `verity_contract` macro: an inductive AST of statements and expressions that the compiler can pattern-match on and transform. The macro generates both representations from a single contract definition, along with kernel-checked bridge facts for generated function bodies [22]. The macro elaborator itself is trusted.

**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.

**What is out of scope.** Verity's correctness guarantees apply to contracts expressible in the supported source fragment (Table 2). The following are explicitly *outside* the current scope:

- **Generic theorem boundary.** The current `SupportedSpec` whole-contract theorem excludes events, custom errors, linked externals/ECMs, fallback/receive entrypoints, low-level calls and returndata, expression-level `keccak256`, and `forEach` loops. Operationally supported uses of these features are compiled and tested, but do not inherit the generic preservation theorem without additional per-contract obligations or trust-surface review.
- **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.

The supported fragment is not yet complete and will be extended in future versions of Verity. Table 2 (Section 5) provides the complete feature matrix, distinguishing features with generic proof coverage from those that are operationally supported and regression-tested but require per-contract proof obligations.

## 4.2. 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 `verity_contract` macro generates both a `Contract` monad value (shallow embedding, directly executable for testing and proof automation) and a `CompilationModel` (deep embedding, the compiler's input) from this single definition. 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 5 shows how each layer transforms this contract, and Section 6 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.

```
verity_contract Counter where
  storage
    count : Uint256 := slot 0

  function increment () : Unit := do
    let current ← getStorage count
    setStorage count (add current 1)

  function decrement () : Unit := do
    let current ← getStorage count
    setStorage count (sub current 1)

  function getCount () : Uint256 := do
    let current ← getStorage count
    return current
```

**Listing 1:** Counter contract in Verity's EDSL. The `verity_contract` macro generates both a monadic `Contract` definition (shallow embedding) and a `CompilationModel` (deep embedding) from this single source.

## 5. The Verity Compiler

Each compilation layer has its own intermediate representation and design trade-offs, which are discussed below. Correctness theorems and the verification methodology are deferred to Section 6.

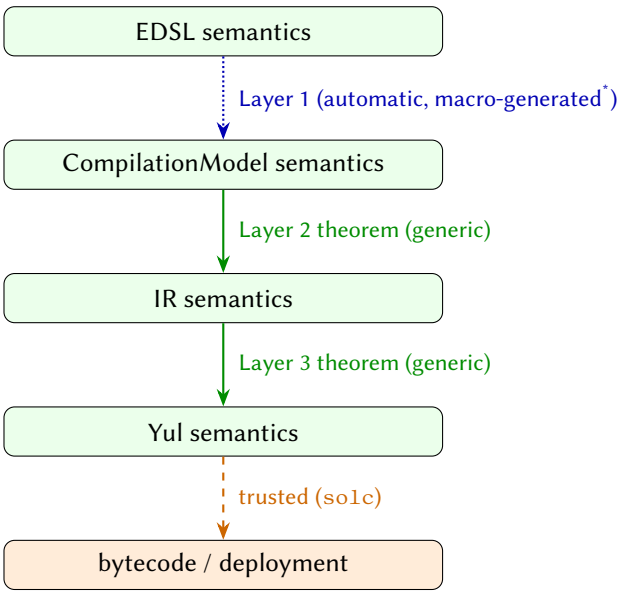


**Figure 1:** Proof structure of Verity's verified compilation chain. Layers 2 and 3 (green solid arrows) are generic theorems proved once for all supported contracts. Layer 1 (blue dotted arrow) is automatic but relies on the `verity_contract` macro elaborator, which is unverified TCB (Section 6.3). The orange dashed arrow marks the trusted `solc` backend.

### 5.1. 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
```

The type Contract $\alpha$ abbreviates `ContractState` $\rightarrow$ `ContractResult` $\alpha$, where `ContractState` models EVM storage (scalar, mapping, array), the transaction sender, message value, block environment, and an append-only event log (see Appendix A for the full definition). The result type `ContractResult`

```
structure CompilationModel where
  name     : String
  fields   : List Field
  constructor : Option ConstructorSpec
  functions : List FunctionSpec
  events   : List EventDef := []
  errors   : List ErrorDef := []
  externals : List ExternalFunction := []

structure FunctionSpec where
  name    : String
  params  : List Param
  returnType : Option FieldType
  isPayable : Bool := false
  body    : List Stmt
```

**Listing 2:** Abbreviated `CompilationModel`: the compiler-facing deep embedding. The macro generates this AST alongside the monadic `Contract` definition.

is an inductive with `success` and `revert` constructors; the monad runner automatically rolls back state on revert.

Layer 1 extracts a `CompilationModel` from EDSL definitions. The `verity_contract` macro generates both the `Contract` monad value (shallow embedding) and the `CompilationModel` (deep embedding) from a single definition. Listing 2 shows the core structure of the deep embedding; the complete pinned definition, including ABI-return metadata, view/pure flags, storage-slot compatibility policy, and local proof obligations, appears in Appendix C.1.

This compiler-facing model enumerates the contract's public functions, their effect on storage, and the correspondence between EDSL computations and model-level interpretations. The extraction is the frontend semantic bridge: the macro emits kernel-checked body-alignment facts between each generated `FunctionSpec` body and the named model body. Full elaborator correctness and whole-contract generation remain trusted.

**Trust model.** Unlike Layers 2 and 3, which are proved by generic theorems, Layer 1 relies on per-contract code generated by the `verity_contract` macro. The macro elaborator is unverified metaprogramming code and is part of the trusted computing base; Section 6.3 discusses its scope, mitigations (bridge theorems, differential tests, macro round-trip fuzzing), and residual risk in detail.

The separate proofs under `Contracts/<Name>/Proofs/` are contract-specification proofs (see Section 6), not compiler-layer proofs.

## 5.2. 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 is kernel-computable via a vendored Keccak-256 engine (Section 6.2). The vendored engine is checked-in Lean source under `Compiler/Keccak/`: `CircuitData.lean` is a generated unrolled/static circuit for Keccak-f[1600], produced by the tracer in `Circuit.lean`, and `Sponge.lean`

```
theorem compile_preserves_semantics
  (model : CompilationModel)
  (selectors : List Nat)
  (hSupported : SupportedSpec model selectors)
  (ir : IRContract)
  (tx : IRTransaction)
  (initialWorld : ContractState)
  (htxNormalized : TxContextNormalized tx)
  (hcalldataSizeFits : TxCalldataSizeFitsEvm tx)
  (hcompile :
   compile model selectors = Except.ok ir) :
  sourceResultMatchesIRResult
   (supportedSourceContractSemantics
    model selectors hSupported tx initialWorld)
   (interpretIR ir tx
    (initialIRStateForTx model tx
     initialWorld))
```

**Listing 3:** Layer 2 preservation theorem (`CompilationModel -> IR`). The `SupportedSpec` witness restricts to the proved fragment; the conclusion relates source semantics and IR execution.

evaluates it inside Lean's kernel. This makes selector derivation a kernel computation. The remaining trust assumption is extensional: this Lean implementation must agree with the EVM's `keccak256` opcode; cross-checks against `solc --hashes` are regression tests, not a proof of opcode agreement.

Internally, `compile` is decomposed into `validateCompileInputs` (input validation) and `compileValidatedCore` (the actual lowering), exposing the validated intermediate state for proof access. The deterministic `--edsl-contract` path invokes this pipeline from a single contract definition to verified Yul output.

**Concrete example.** To make the lowering tangible, consider the counter's `increment` function from Section 4.2. Layer 1 produces a `FunctionSpec` whose body is the statement list `[letVar "current" (storage "count"), setStorage "count" (add (localVar "current") (literal 1))]`, a direct AST mirror of the EDSL source. Layer 2 compiles this to an `IRFunction` whose body is a list of `IRStmt` (which are `YulStmt` by the alias in Section 5.3):

```
-- IR for Counter.increment (simplified)
[let current := sload(0),
 sstore(0, add(current, 1)),
 return(0, 0)]
```

The deep-embedding `storage "count"` has become `sload(0)` (the slot number resolved at compile time), and `setStorage` has become `sstore`. The proof obligation is that interpreting the source statement list on the `CompilationModel` evaluator agrees with executing the IR statements on the IR evaluator, for every initial state.

**The Layer 2 preservation theorem.** Listing 3 shows the generic Layer 2 theorem. It states that for *any* compilation model satisfying the `SupportedSpec` witness, successful compilation produces an IR contract whose execution matches the source-level semantics.
The `sourceResultMatchesIRResult` relation requires agreement on success/revert status, returned data, final storage, and emitted events. The preconditions `htxNormalized` and `hcalldataSizeFits` are lightweight structural conditions on the transaction (fields initialized to their transaction values, calldata length within EVM bounds).

**The SupportedSpec witness.** The generic Layer 2 theorem is parameterized by a `SupportedSpec` witness that restricts the model to the currently proved fragment:

```
structure SupportedSpec (spec : CompilationModel)
    (selectors : List Nat) where
  invariants : SupportedSpecInvariants spec selectors
  surface : SupportedSpecSurface spec
  functions : ∀ fn, fn ∈ spec.functions →
            SupportedFunction spec fn
```

The `SupportedSpecSurface` restricts the contract-level shape for the current generic theorem boundary, while `SupportedFunction` restricts each function body to the proved expression and statement forms. Each function must satisfy `SupportedFunction` (supported parameter types and body within the `ExprCompileCore` and `SupportedStmtList` witnesses). The fragment now covers storage-heavy contracts with mapping/struct reads and writes, if/else branching (with terminal branches), signed and fixed-point arithmetic, storage arrays, and memory/transient storage operations. Events, custom errors, and forEach loops remain outside the generic theorem and are 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 2) expands as additional statement forms are proved.

### 5.3. Layer 3: IR to Yul

The final verified stage emits Yul. A key design choice is that the IR and Yul share the same AST types:

```
abbrev IRStmt := Yul.YulStmt
abbrev IRExpr := Yul.YulExpr
```

This means the "lowering" from IR to Yul is not a translation between different representations but rather an assembly of per-function IR bodies into a complete `YulObject` with dispatch logic, deploy code, and ABI encoding. The code generator produces:

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

The Layer 3 proof uses Verity's own Yul interpreter. For all pure builtins, bridge theorems prove agreement with EVMYulLean [13], an independently maintained Lean 4 formalization validated against the official Ethereum conformance test suite. The generated Yul is subsequently compiled to EVM bytecode by `solc` (v0.8.33), which lies outside the verified boundary.

### 5.4. Supported Source Fragment

Table 2 summarizes each EDSL feature and distinguishes *operational* support (the feature is compiled and regression-tested) 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; the remaining forms are addressed in future work.

Formally, the proved fragment is characterized by the `SupportedSpec` predicate in the Layer 2 theorem. The predicate decomposes into surface-specific sub-predicates (`CoreSurface`, `StateSurface`, `CallSurface`, `EffectSurface`) that each enumerate which AST constructors are within scope. Statement-level if/else is supported when both branches are *terminal* (every path ends with return, stop, or revert). Storage reads for mappings, structs, and arrays are proved when let-bound in statement position. A "Tier 2" variant (`SupportedSpecExceptMappingWrites`) relaxes the state surface to admit mapping and struct writes while keeping all other boundaries unchanged. It is consumed by separate whole-contract theorems such as `compile_preserves_semantics_except_mapping_writes`; the primary `compile_preserves_semantics` theorem does not depend on this relaxed variant. Appendix D gives the proof-interface decomposition used by both variants.

**Table 2**
EDSL feature support. "Operational" means the feature is compiled and regression-tested. "Generic proof" indicates coverage by the `SupportedSpec` Layer 2 theorem. Features outside the generic proof are still compiled, tested, and available for per-contract reasoning.

| Feature | Operational | Generic proof |
|---|---|---|
| Core arithmetic, comparisons, booleans | supported | proved |
| Signed arithmetic (sdiv, smod, slt, sgt, sar) | supported | proved |
| Bitwise (and/or/xor/not, shifts), min/max, ceilDiv | supported | proved |
| Fixed-point math (wMulDown, wDivUp, mulDiv) | supported | proved |
| Local variables, conditional expressions (ite) | supported | proved |
| Require/revert, return, stop | supported | proved |
| Storage read/write (uint256 and address fields) | supported | proved |
| Mapping/struct reads (let-bound in statements) | supported | proved |
| Mapping/struct writes | supported | proved (Tier 2) |
| Storage array ops (push, element, length) | supported | proved |
| If/else branching (terminal branches) | supported | proved |
| Selector dispatch + ABI loading | supported | proved |
| Payable modifier | supported | proved |
| Linear memory (mstore/mload), transient storage (tstore/tload) | supported | proved[†] |
| calldataload, calldatasize, blobbasefee | supported | proved |
| Internal helper calls | supported | compositional |
| forEach loops | supported | not in generic proof |
| Events (emit), raw log, event ABI parity | supported | excluded |
| Custom errors, multi-return, returnValues | supported | excluded |
| Linked externals, ECM | supported | excluded |
| Constructor, fallback/receive | supported | excluded |
| Low-level calls, delegatecall, returndata | supported | not in generic proof |
| keccak256 (expression-level) | supported | not in generic proof |
| ABI JSON artifact generation | supported | n/a |
| ETH balance tracking, create/create2 | not yet implemented | |

[†]Transient storage operations (EIP-1153) are proved at the statement level within the generic theorem; they are excluded from the cross-layer `resultsMatch` relation because transient storage resets at transaction boundaries (Section 6.1).

Table 3 summarizes the verification status of each layer. Both the mapping-slot derivation and function-selector computation are now kernel-computable (Section 6.2). The final stage trusts `solc`.

# 6. Formal Verification of the Compilation Chain

## 6.1. Correctness Theorems

The central correctness property states that compiling through the kernel-checked frontend bridges and the two generically proved compiler stages produces the same observable behavior as executing the source contract directly (recall Figure 1).

Each box in Figure 1 is a concrete, executable Lean interpreter. The EDSL semantics is the `Contract` monad itself (`ContractState → ContractResult`); the CompilationModel semantics is an inductive evaluator (`evalExpr`/`execStmt` in `SourceSemantics.lean`) over a `RuntimeState` that pairs variable bindings with a `ContractState`; the IR semantics is an analogous evaluator (`IRInterpreter.lean`) over `IRState`; and the Yul semantics is provided by Verity's own Yul interpreter, whose pure-arithmetic builtins (add, sub, mul, div, mod, comparisons, bitwise operations) are proven equivalent to EVMYulLean [13], an independently maintained Lean 4 formalization validated against the official Ethereum conformance test suite. This bridge gives the compilation proofs an

**Table 3**
Verification layers in the Verity compilation pipeline.

| Layer | Stage | Status | Axioms |
|---|---|---|---|
| 1 | EDSL -> CompilationModel | Automatic (per-contract bridges) | none |
| 2 | CompilationModel -> IR | Generic (supported fragment) | none |
| 3 | IR -> Yul | Generic | none |
| ext | Yul -> bytecode (solc) | Trusted (0.8.33+commit.64118f21) | N/A (unverified) |

**Table 4**
Status and generality of Verity's main correctness claims.

| Claim layer | Generality | What is mechanized |
|---|---|---|
| Source → `CompilationModel` | Per-contract generated bridges; macro trusted | Kernel-checked function-body bridge equalities generated by `verity_contract`; full macro elaboration and contract structure remain TCB |
| `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 |

externally auditable semantic reference for all pure EVM operations. Because all four interpreters are ordinary Lean functions, they are directly testable: the differential-testing suite (Section 7) runs each against `solc`-compiled bytecode as an independent cross-check.

The argument has two levels of generality: the middle and backend compiler theorems are generic, while the source-level connection is made by macro-emitted bridge facts that Lean checks for contracts in the supported source fragment identified in Table 2.

We distinguish two kinds of proof in Verity. *Compiler-layer proofs* establish generic preservation for Layers 2–3, with Layer 1 connected by macro-emitted, kernel-checked bridge facts and a trusted macro elaborator. *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: generic compiler proofs apply to the supported compiler fragment, 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, persistent storage state (both scalar slots and mapping slots), and emitted events. Gas costs and transient storage (EIP-1153) are excluded from the comparison. Transient storage resets at transaction boundaries, so it does not affect cross-transaction observable state; within a single transaction, any influence of transient storage on return values or persistent writes is captured indirectly through the returned-data and storage components of `resultsMatch`. Adding a direct transient-storage comparison is a planned extension as stateful-builtin coverage widens. Event logs are compared by `resultsMatch`; gas-sensitive behavior is instead validated operationally by differential testing against `solc`-compiled bytecode.

Table 4 summarises what kind of result Verity provides. The strongest generic mechanized result is the compiler core from `CompilationModel` to Yul. The source-level bridge is intentionally stated as per-contract generated facts rather than as a proof of the macro elaborator over all EDSL programs.

**Layer 3 preservation theorem.** Listing 4 shows the Layer 3 preservation theorem used to connect IR semantics to emitted Yul semantics for any transaction and initial state. The helper `initialState.withTx tx` copies sender, value, addresses, and block context from the transaction

```
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)
  (hNoHasSelector : forall fn,
   fn in contract.functions ->
    yulStmtsNoRef "__has_selector" fn.body)
  (hHasSelectorDead : forall fn,
   fn in contract.functions ->
    HasSelectorDeadBridge fn.body)
  (hLoopFree : forall fn,
   fn in contract.functions ->
    yulStmtsLoopFree fn.body = true)
  (hbody : forall fn, fn in contract.functions ->
   resultsMatch
    (execIRFunction fn tx.args
       (initialState.withTx tx))
    (interpretYulBody fn tx
       (initialState.withTx tx))) :
  resultsMatch
   (interpretIR contract tx initialState)
   (interpretYulFromIR contract tx initialState)
```

**Listing 4:** Layer 3 preservation theorem (`IR -> Yul`). `withTx` injects transaction fields into the initial state (Appendix E.1).

into the initial state (see Appendix E.1 for the full definition).
Beyond the selector bound, the theorem requires several 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; `hdispatchGuardSafe` ensures per-function dispatch guard safety; `hNoHasSelector` and `hHasSelectorDead` ensure the dispatch bridge is sound (no selector collisions); and `hLoopFree` requires the function bodies to be loop-free (matching the current supported fragment). The `hbody` hypothesis is discharged by `ir_function_body_equiv`, which derives per-function IR/Yul agreement from statement-level equivalence and a fuel-adequacy lemma. Appendix E unpacks these hypothesis groups against the current theorem in the codebase.

**Backend composition spine.** The codebase contains a reusable Layers 2–3 composition theorem for the backend side. The theorem displayed below is intentionally narrower than the paper-level source-to-Yul guarantee: paper-level composition also uses the source-to-IR relation supplied separately by `compile_preserves_semantics` over `SupportedSpec`. This theorem relates IR execution to generated-Yul execution for a successfully compiled IR contract under the Layer 3 structural hypotheses. Concrete instantiations discharge the theorem's structural, dispatch, loop-freedom, and parameter-erasure hypotheses for individual contracts. For example, `counter_supported_spec_compile_preserves_semantics` instantiates the separate Layer 2 source-to-IR theorem for the counter contract from Section 4.2.

```
theorem layers2_3_ir_matches_yul
  (spec : CompilationModel)
  (selectors : List Nat)
  (irContract : IRContract)
  (tx : IRTransaction) (initialState : IRState)
  (hCompile :
   compile spec selectors = .ok irContract)
  (hselector : tx.functionSelector
   < selectorModulus)
  (hNoWrap : 4 + tx.args.length * 32
   < evmModulus)
  (hvars : initialState.vars = [])
  (hmemory : initialState.memory = fun _ ⇒0)
  (htransient :
   initialState.transientStorage = fun _ ⇒0)
  (hreturn : initialState.returnValue = none)
  (hparamErase : forall fn,
   fn in irContract.functions ->
    paramLoadErasure fn tx
     (initialState.withTx tx))
  (hdispatchGuardSafe : forall fn,
   fn in irContract.functions ->
    DispatchGuardsSafe fn tx)
  (hNoHasSelector : forall fn,
   fn in irContract.functions ->
    yulStmtsNoRef "__has_selector" fn.body)
  (hHasSelectorDead : forall fn,
   fn in irContract.functions ->
    HasSelectorDeadBridge fn.body)
  (hLoopFree : forall fn,
   fn in irContract.functions ->
    yulStmtsLoopFree fn.body = true)
  (hWF : ContractWF irContract)
  (hNoFallback :
   irContract.fallbackEntrypoint = none)
  (hNoReceive :
   irContract.receiveEntrypoint = none) :
  resultsMatch
   (interpretIR irContract tx initialState)
   (interpretYulFromIR irContract tx
    initialState)
```

**Listing 5:** IR-to-Yul backend composition spine. Given a successfully compiled model and the Layer 3 structural hypotheses, IR execution matches generated-Yul execution.

**Contract-specification proofs.** Contract-specification proofs are a separate concern from compiler proofs. Where the theorems above show that the *compiler preserves semantics*, specification proofs show that the *EDSL implementation meets its human-written specification*. For example, the counter contract has 9 contract-specific theorems including:

```
theorem increment_getCount_meets_spec (s : ContractState) :
 let s' := ((increment).run s).snd
 let result := ((getCount).run s').fst
 increment_getCount_spec s result

theorem decrement_at_zero_wraps_max
  (s : ContractState) (h : s.storage 0 = 0) :
```

```
let s' := ((decrement).run s).snd
s'.storage 0 = EVM.MAX_UINT256
```

These proofs proceed by unfolding definitions, simplification, and reflexivity, since the EDSL execution reduces to definitionally equal terms. Any EDSL contract inherits the compilation-correctness guarantee from the generic compiler theorems; only the specification proofs are contract-specific.

## 6.2. Project Axioms

At the pinned snapshot, Verity uses 0 project-specific `axiom` declarations beyond Lean's built-in foundational axioms (`propext`, `Quot.sound`, `Classical.choice`). All compiler-core correctness facts for Layers 2–3, and the generated Layer 1 bridge facts, are proved theorems with no additional `axiom` declarations. Note that this zero-axiom property is a statement about the Lean proof system: the operational trusted computing base (Table 5) remains non-trivial and includes components such as `solc`, the macro elaborator, and the Lean kernel itself.

Both Keccak-dependent proof obligations have been internalized as theorems:

**`keccak256_first_4_bytes`.** EVM function-selector computation is now kernel-computable via a vendored Keccak-256 engine under `Compiler/Keccak/`. The checked-in Lean source includes `CircuitData.lean`, a generated unrolled/static Keccak-f[1600] circuit produced from `Circuit.lean`, and runs inside Lean's kernel evaluator.

**`solidityMappingSlot_lt_evmModulus`.** The bound on mapping-slot derivations ($< 2^{256}$) is now proved structurally: the Keccak engine's output-length bound (`squeeze256_size`) and a big-endian conversion lemma (`fromByteArrayBigEndian_lt_of_size`) yield the result as a theorem.

The Layer 3 dispatch bridge uses explicit per-contract theorem hypotheses (`HasSelectorDeadBridge`, `DispatchGuardsSafe`), discharged at proof time for each contract. As a result, the proof-system TCB contains only Lean's three standard foundational axioms, with all Keccak-256 computations kernel-computable. Keccak-256 is still trusted at the operational level (the kernel implementation must match the EVM's keccak256 opcode). CI cross-checks against `solc --hashes` and end-to-end regression suites are regression tests for this assumption, not a mechanized proof of the Keccak permutation or opcode agreement.

## 6.3. Trust Boundary

Verity enumerates precisely what is proved and what is assumed. Inside the verified compiler core, proof checking relies on Lean's kernel. Operationally, trust also extends to the macro elaborator, semantic reference assumptions, Keccak agreement with EVM `keccak256`, `solc` for Yul-to-bytecode, gas, and any linked libraries, ECMs, or local obligations used by a contract.

**What is proved.** The `CompilationModel`-to-Yul compiler core (Layers 2–3) is mechanically verified within Lean's type theory (see Section 6.2 above for the axiom status). Every contract in the supported fragment inherits the generic compiler-correctness guarantee for these stages. Layer 1 contributes macro-emitted, kernel-checked bridge facts, with the macro elaborator itself trusted.

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

**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.

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

| Component | Role | Trust Type |
|---|---|---|
| Solidity Compiler (solc) | Compiles Yul → EVM bytecode. | external tool |
| Keccak-based Selector Computation | Function selector derivation (bytes4(keccak256(signature))). | operational |
| 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 |

**EVM/Yul semantics.** Verity implements its own Yul interpreter (`evalBuiltinCallWithContext`). For all pure builtins (arithmetic, comparisons, bitwise operations), 15 bridge theorems in `EvmYulLeanBridgeLemmas.lean` prove that Verity's semantics agree with EVMYulLean [13], an independently maintained formalization validated against the official Ethereum conformance test suite. State-dependent builtins (sload, caller, calldataload, etc.) are modeled internally. Gas consumption is not modeled; semantic correctness does not imply gas-bounded liveness.

**Macro elaborator (`verity_contract`).** Generates both the EDSL `Contract` monad value and the `CompilationModel` from a single syntax tree. The EDSL definition is the authoritative source for contract behavior; the macro is the authoritative elaboration boundary between the user-facing DSL and the compiler-facing AST. A bug in this unverified metaprogramming code could cause the two representations to diverge. Macro-generated bridge theorems use Lean's kernel-checked definitional equality to prove function-body identity, and 520 differential tests plus macro round-trip fuzzing provide additional coverage. The elaborator itself remains an unverified trusted component, as bridge theorems cover function bodies but not the full contract structure (constructor, dispatch, error encoding).

**Linked Yul libraries.** External functions injected at compile time (e.g., Poseidon hash) are outside the generic proof and must be audited separately. The compiler validates names and arities, and the `--deny-unchecked-dependencies` flag fails compilation if any linked external has not been audited.

Table 5 lists the full TCB. Beyond the core components above, it includes: *External Call Modules (ECMs)*, reusable typed patterns for calling external contracts (ERC-20, ERC-4626, oracle reads, precompiles) whose interface assumptions are documented and auditable; and *LocalUnsafe/Refinement Obligations*, which let a function declare a localized proof obligation for an unsafe boundary without marking the whole contract as opaque.

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's monadic interpreter can expose intermediate state in reverted computations, although `Contract.run` applies rollback at the call boundary, matching EVM behavior at that level.

# 7. Evaluation

The evaluation covers four dimensions: contract coverage, the split between generic and contract-specific proof work, verification effort per contract, and the role of testing.

## 7.1. Compiler Infrastructure (Generic)

The compiler proof infrastructure is proved once and applies to every contract in the supported fragment. Table 2 (Section 5) lists the EDSL features that are operationally supported and distinguishes those covered by the generic proof.

**Scope of the supported fragment.** The generic `SupportedSpec` theorem covers the core computational fragment: storage-heavy contracts with mappings, structs, arrays, signed and fixed-point arithmetic, conditional branching, internal helpers, and access control. Features outside this fragment (events, custom errors, forEach loops, linked externals) are operationally supported and covered by differential testing but require per-contract proof obligations rather than inheriting the generic guarantee. This is an important scope limitation: events are used by virtually every production contract. Full EDSL coverage is future work; the current boundary is precisely characterized by the `SupportedSpecSurface` and `SupportedFunction` predicates.

**Proof effort breakdown.** A `grep`-level scan of the 259 Lean source files at the pinned snapshot counts 1 565 `theorem` declarations, none using `sorry` or `admit`. Of these, roughly 1 006 are compiler-infrastructure lemmas proved once (324 compilation, 643 IR generation, 39 Yul generation) and 221 are standard-library proofs. The remaining 277 are contract-specific specification theorems across 11 contract families. A developer adding a new contract pays only the specification-theorem cost; the compiler infrastructure is inherited.

## 7.2. Contract-Level Verification

**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. All families compile and pass differential testing. The proof-complete generic-compiler examples are the contracts that stay inside `SupportedSpec`, such as counters, simple storage, ledger, and owned contracts without excluded surfaces. ERC-20, ERC-721, Vault, hash, and external-call examples exercise important operational surfaces, but features outside the generic theorem boundary serve as testing and trust-reporting evidence rather than widening that boundary.

**Verification effort per contract.** Adding a new contract requires writing the EDSL definition and its specification proofs; compiler correctness is inherited for the supported fragment. To give a sense of scale: Ledger is 40 lines of contract source, 55 lines of specification, and 682 lines of proof (a 17:1 proof-to-source ratio) yielding 33 specification theorems. SimpleToken is 71/49/1 062 lines (15:1) yielding 61 theorems. SimpleStorage, the smallest verified contract at 19 source lines, requires 229 lines of proof (12:1) for 20 theorems. Most proof lines are tactic scripts; the developer writes the theorem *statements* and Lean's automation discharges the bulk of the obligations. These ratios should be read as early-stage verification engineering cost, not as a direct measure of proof complexity or as a fixed per-contract tax. The compiler infrastructure is amortized across supported contracts; new contracts mainly require

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

| Contract | Source-level guarantees | 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`. | Combines access control, arithmetic side conditions, and coupled balance/supply reasoning. | Exact per-pair balance equations; no global sum invariant. |

**Table 7**
Claim-level evidence map. Each headline claim is backed by a mechanized proof artifact, a test artifact, or both, with the relevant TCB assumption noted.

| Claim | Proof artifact | Test artifact | TCB assumption |
|---|---|---|---|
| Zero project axioms | CI axiom scan; `AXIOMS.md` | n/a | Lean kernel |
| Layer 2 correctness | `compile_preserves_semantics` | Foundry diff-tests | Macro elaborator |
| Layer 3 correctness | `yulCodegen_preserves_semantics` | Foundry diff-tests | Yul interpreter |
| End-to-end (L2+L3) | `layers2_3_ir_matches_yul` | n/a | Macro + Lean kernel |
| EVMYulLean bridge | 15 bridge lemmas | EVMYulLean conformance tests | Pure builtins only |
| Contract specs | Per-contract theorems | Foundry property tests | Macro + `solc` |
| Yul → bytecode | n/a | `solc` compilation | `solc` (trusted) |

human-written specifications and contract-specific proof scripts. Contracts using features outside the generic theorem boundary also incur local obligations or trust-surface review.

## 7.3. Testing Infrastructure

**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.** Verity proves that successful `deposit`, `withdraw`, `transfer`, and `getBalance` agree with their specifications on observable storage, that insufficient-balance calls preserve revert behavior, and that transfers conserve sender/recipient balance sums while leaving unrelated accounts unchanged.

**SimpleToken case study.** `SimpleToken` adds access control and supply accounting. Verity proves that owner-authorized `mint` updates the beneficiary balance and `totalSupply` as specified, that non-owner `mint` reverts, and that transfers preserve `totalSupply`. Balance conservation is stated as exact per-pair equations rather than a global sum invariant (which is false when the address list contains duplicates).

**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.

## 8. Limitations and Future Work

**Scope of the EDSL.** The supported source fragment is described in Section 4.1. 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. Verity's own Yul interpreter is the semantic authority, but its pure-builtin semantics are proven equivalent to EVMYulLean [13], which is validated against the official Ethereum conformance test suite. The remaining trust assumption is `solc`'s fidelity to those same semantics. Version pinning and differential testing provide additional mitigations.

**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.

**Keccak-256 trust.** Keccak-256 computations (function selectors, mapping-slot derivations) are kernel-computable via a vendored Keccak engine (Section 6.2). The vendored code is checked-in Lean source under `Compiler/Keccak/`, including the generated `CircuitData.lean` unrolled/static Keccak-f[1600] circuit. Its correctness relative to the EVM's `keccak256` opcode is trusted operationally. Cross-checks against `solc --hashes` and end-to-end regression suites are tests for that assumption, not proofs. Formally verifying the Keccak-256 permutation in Lean's logic remains 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`. The Lean kernel is itself part of the trusted computing base: a soundness bug in Lean's type checker would invalidate all mechanized proofs. *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.

## 9. Conclusion

Verity provides a three-layer verified compiler for EVM smart contracts, implemented as a Lean 4 EDSL whose compilation chain to Yul is mechanically verified. Layers 2 and 3 are proved generically and Layer 1 is verified per-contract via macro-generated bridge theorems. The trust boundary (Section 6.3) separates what is proved from what is assumed, and an evaluation on 11 contract families shows that the approach scales to storage-heavy contracts with mappings, access control, and arithmetic, while enabling contract-specific correctness arguments (e.g., balance conservation, access-control enforcement) that compose with the compiler guarantee.

The generic compiler proof currently covers a core computational fragment; events and loops are operationally supported and regression-tested but require per-contract proof obligations. Extending the proved fragment toward full EDSL coverage, reducing the trusted boundary around the Yul-to-bytecode backend, and formally verifying the vendored Keccak engine are the primary directions for future work. More broadly, Verity suggests that Lean 4's combination of a dependently typed programming language and an interactive theorem prover provides a productive foundation for verified smart-contract compilation, an approach that may generalize to other blockchain targets beyond the EVM.

## Data availability statement

The Verity compiler, all Lean proof sources, and the Foundry test suite are available as open-source software at https://github.com/th0rgal/verity. The companion paper repository is available at https://github.com/th0rgal/verity-paper and contains a reproducibility script (`make regen && make check`) that regenerates every table and cross-checks all metrics reported in this paper. All claims are pinned to the Verity artifact commit `8c58fb2a8f18`.

## Declaration on Generative AI

During the preparation of this work and its accompanying artifact, the author used Claude (Anthropic) and OpenAI Codex as AI-assisted programming and writing tools. These tools were used to support the development, refactoring, and debugging of Lean proof scripts in the Verity source code; to suggest proof strategies and intermediate lemmas; to help inspect proof obligations and error messages; to assist with checking consistency between manuscript claims and the implementation; and to copyedit manuscript prose for grammar, clarity, and concision.

All scientific claims, theorem statements, definitions, implementation changes, and manuscript text were reviewed, edited, and validated by the human author. The Lean proofs and code included in the artifact were accepted by the Lean kernel and the project test suite, but the author does not treat AI-generated suggestions as evidence of correctness. The author takes full responsibility for the correctness, originality, integrity, and final content of the paper and artifact.

## Acknowledgments

The author thanks Denisa Diaconescu for her guidance during the ETHReS submission process and for detailed feedback on the manuscript. The author thanks Charles Guillemet and Louis Milhaud for reviewing the paper and providing helpful feedback. The author also thanks Benjamin Flores for contributions to the Verity source code and discussions about the artifact.

## References

[1] P. Daian, Analysis of the DAO exploit, https://web.archive.org/web/20240107174937/https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016. Accessed: 2026-04-15.

[2] Vyper Team, Incorrectly allocated named re-entrancy locks, https://github.com/vyperlang/vyper/security/advisories/GHSA-5824-cm3x-3c38, 2023. Accessed: 2026-04-15.

[3] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: Principles of Security and Trust (POST 2017), volume 10204 of *LNCS*, Springer, Berlin, Heidelberg, 2017, pp. 164–186. doi:10.1007/978-3-662-54455-6_8.

[4] J. Feist, G. Grieco, A. 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), IEEE, Montreal, QC, Canada, 2019, pp. 8–15. doi:10.1109/WETSEB.2019.00008.

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

[6] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, G. Roşu, KEVM: A complete formal semantics of the Ethereum Virtual Machine, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, Oxford, UK, 2018, pp. 204–217. doi:10.1109/CSF.2018.00022.

[7] Certora, Certora prover documentation, https://docs.certora.com/, 2024. Accessed: 2026-04-15.

[8] X. Leroy, Formal verification of a realistic compiler, Communications of the ACM 52 (2009) 107–115. doi:10.1145/1538788.1538814.

[9] R. Kumar, M. O. Myreen, M. Norrish, S. Owens, CakeML: A verified implementation of ML, in: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, New York, NY, USA, 2014, pp. 179–191. doi:10.1145/2535838.2535841.

[10] J. Avigad, L. Goldberg, D. Levit, Y. Seginer, A. Titelman, A proof-producing compiler for blockchain applications, in: 14th International Conference on Interactive Theorem Proving (ITP 2023), volume 268 of *LIPIcs*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 7:1–7:19. doi:10.4230/LIPIcs.ITP.2023.7.

[11] L. de Moura, S. Ullrich, The Lean 4 theorem prover and programming language, in: Automated Deduction – CADE 28, volume 12699 of *LNCS*, Springer, Cham, 2021, pp. 625–635. doi:10.1007/978-3-030-79876-5_37.

[12] Solidity Team, Solidity documentation, https://docs.soliditylang.org/, 2024. Accessed: 2026-04-15.

[13] NethermindEth, EVMYulLean: EVM and Yul semantics in Lean 4, https://github.com/NethermindEth/EVMYulLean, 2025. Accessed: 2026-04-15.

[14] A. W. Appel, Verified software toolchain, in: Programming Languages and Systems – ESOP 2011, volume 6602 of *LNCS*, Springer, Berlin, Heidelberg, 2011, pp. 1–17. doi:10.1007/978-3-642-19718-5_1.

[15] J. Avigad, L. Goldberg, D. Levit, Y. Seginer, A. Titelman, A proof-producing compiler for blockchain applications, Journal of Automated Reasoning 69 (2025) 9. doi:10.1007/s10817-025-09723-y.

[16] G. Roşu, T. F. Şerbănuţă, An overview of the K semantic framework, Journal of Logic and Algebraic Programming 79 (2010) 397–434. doi:10.1016/j.jlap.2010.03.012.

[17] V. Sjöberg, Y. Sang, S.-C. Weng, Z. Shao, DeepSEA: A language for certified system software, Proceedings of the ACM on Programming Languages 3 (2019) 1–27. doi:10.1145/3360562.

[18] V. Sjöberg, K. Dave, D. Britten, M. A. Schett, X. Sun, Q. Wang, S. N. Anderson, S. Reeves, Z. Shao, Foundational verification of smart contracts through verified compilation, arXiv preprint arXiv:2405.08348, 2024. doi:10.48550/arXiv.2405.08348. arXiv:2405.08348.

[19] D. Annenkov, J. B. Nielsen, B. Spitters, ConCert: A smart contract certification framework in Coq, in: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), ACM, 2020, pp. 215–228. doi:10.1145/3372885.3373829.

[20] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, J. Tesson, Mi-Cho-Coq, a framework for certifying Tezos smart contracts, in: Formal Methods. FM 2019 International Workshops, volume 12232 of *LNCS*, Springer, Cham, 2019, pp. 368–379. doi:10.1007/978-3-030-54994-7_28.

[21] Mathlib Community, Mathlib4: The Lean 4 mathematical library, https://github.com/leanprover-community/mathlib4, 2024. Accessed: 2026-04-15.

[22] J. Svenningsson, E. Axelsson, Combining deep and shallow embedding for EDSL, in: Trends in Functional Programming (TFP 2012), volume 7829 of *LNCS*, Springer, 2013, pp. 21–36. doi:10.1007/978-3-642-40447-4_2.

# A. EDSL Core Types

This appendix reproduces the code shapes that support the main text. The snippets are taken from the pinned Verity artifact commit and are lightly formatted for width. Nonessential deriving clauses and comments are omitted only where stated.

## A.1. Contract State and Result

The `Contract` monad is a state-passing function over `ContractState`. Reverts carry a state value internally, but the contract runner applies rollback at the call boundary.

```
structure ContractState where
  storage     : Nat → Uint256
  transientStorage : Nat → Uint256
  storageAddr : Nat → Address
  storageMap  : Nat → Address → Uint256
  storageMapUint : Nat → Uint256 → Uint256
  storageMap2 : Nat → Address → Address → Uint256
  storageArray : Nat → List Uint256
  sender      : Address
  thisAddress : Address
  msgValue    : Uint256
  blockTimestamp : Uint256
  blockNumber : Uint256 := 0
  chainId     : Uint256 := 0
  blobBaseFee : Uint256 := 0
  calldataSize : Uint256 := 0
  calldata    : List Nat := []
  memory      : Nat → Uint256 := fun _ ⇒0
  knownAddresses : Nat → FiniteAddressSet
  events      : List Event := []

inductive ContractResult (α : Type) where
  | success : α → ContractState → ContractResult α
  | revert : String → ContractState → ContractResult α

abbrev Contract (α : Type) :=
  ContractState → ContractResult α
```

Listing 6: Core EDSL state and result types.

# B. The `verity_contract` Macro

The macro is the Layer 1 trusted elaboration boundary. It accepts one contract definition and emits both the shallow EDSL definitions and the deep `CompilationModel` consumed by the compiler.

## B.1. Accepted Syntax

```
syntax (name := verityContractCmd)
  "verity_contract " ident " where "
  "storage " verityStorageField*
  ("errors " verityError+)?
  ("constants " verityConstant+)?
  ("immutables " verityImmutable+)?
  ("linked_externals " verityExternal+)?
  (verityConstructor)?
```

```
  (veritySpecialEntrypoint)*
  verityFunction* : command
```

Listing 7: Syntax of the `verity_contract` command macro.

## B.2. Elaboration Pipeline

The elaborator parses and validates the syntax, opens a namespace named after the contract, emits storage and constant definitions, emits each function's EDSL body and model body, emits bridge theorems, and then emits the whole contract `spec`. The final loops add index simplification lemmas and the historical `_semantic_preservation` aliases.

```
@[command_elab verityContractCmd]
def elabVerityContract : CommandElab := fun stx =>do
 let (contractName, fields, errorDecls, constDecls,
     immutableDecls, externalDecls, ctor, functions) <-
  parseContractSyntax stx

 validateGeneratedDefNamesPublic fields constDecls functions
 validateConstantDeclsPublic constDecls
 validateImmutableDeclsPublic fields constDecls immutableDecls ctor
 validateExternalDeclsPublic externalDecls
 validateFunctionDeclsPublic
  fields errorDecls constDecls immutableDecls
  externalDecls ctor functions

 elabCommand (<- `(namespace $contractName))
 try
  for constant in constDecls do
   elabCommand (<- mkConstantDefCommandPublic constant)

  for field in fields do
   elabCommand (<- mkStorageDefCommandPublic field)

  for imm in immutableDecls.zipIdx do
   elabCommand (<-
    mkStorageDefCommandPublic
     (immutableStorageFieldDecl fields imm.1 imm.2))

  for fn in functions do
   let fnCmds <- mkFunctionCommandsPublic
    fields constDecls immutableDecls functions fn
   for cmd in fnCmds do
    elabCommand cmd
   elabCommand (<- mkBridgeCommand fn.ident)

  elabCommand (<- mkSpecCommandPublic
   (toString contractName.getId) fields errorDecls
   constDecls immutableDecls externalDecls ctor functions)

  for cmd in (<- mkFindIdxFieldSimpCommandsPublic
     contractName fields) do
   elabCommand cmd

  for cmd in (<- mkFindIdxParamSimpCommandsPublic
     contractName ctor functions) do
   elabCommand cmd

```

```
    for fn in functions do
      elabCommand (<- mkSemanticBridgeCommand contractName fields fn)

    elabCommand (<- `(end $contractName))
  catch err ⇒
    elabCommand (<- `(end $contractName))
    throw err
```

Listing 8: Layer 1 macro elaboration pipeline from `Verity/Macro/Elaborate.lean`.

### B.3. Generated Bridge Theorems

For a function `increment`, the macro emits names of the following form: `increment`, `increment_modelBody`, `increment_model`, and `increment_bridge`. The bridge theorem is kernel-checked by reflexivity and proves that the `FunctionSpec` body generated for the compiler equals the separately named model body.

```
theorem increment_bridge :
  (Compiler.CompilationModel.FunctionSpec.body
    (increment_model :
      Compiler.CompilationModel.FunctionSpec)) =
  increment_modelBody := rfl
```

Listing 9: Shape of the generated function-body bridge theorem.

The macro also emits `increment_semantic_preservation` as a historical alias; it is definitionally identical and omitted here. This bridge covers function-body alignment. It does not by itself verify the whole macro elaborator, constructor generation, dispatch shape, or error encoding. Those remain part of the Layer 1 trusted boundary described in Section 6.3.

## C. Compilation Pipeline Types

### C.1. `CompilationModel`

The main paper shows an abbreviated `CompilationModel`. The current pinned definition includes ABI return metadata, view/pure flags, internal helper markers, and local trust obligations.

```
structure FunctionSpec where
  name : String
  params : List Param
  returnType : Option FieldType
  returns : List ParamType := []
  isPayable : Bool := false
  isView : Bool := false
  isPure : Bool := false
  body : List Stmt
  isInternal : Bool := false
  localObligations : List LocalObligation := []

structure ConstructorSpec where
  params : List Param
  isPayable : Bool := false
  body : List Stmt
  localObligations : List LocalObligation := []

structure CompilationModel where
  name : String
```

```
  fields : List Field
  reservedSlotRanges : List ReservedSlotRange := []
  slotAliasRanges : List SlotAliasRange := []
  constructor : Option ConstructorSpec
  functions : List FunctionSpec
  events : List EventDef := []
  errors : List ErrorDef := []
  externals : List ExternalFunction := []
```

Listing 10: Full current `FunctionSpec`, `ConstructorSpec`, and `CompilationModel` field inventory, with deriving clauses omitted.

## C.2. IR and Yul AST

Verity's IR is a disciplined use of the Yul AST. Layer 2 produces an `IRContract`; Layer 3 assembles those statements into deploy and runtime Yul code. This avoids a separate AST translation between IR and Yul, but still leaves semantic work: dispatch code, ABI loading, and function-body execution must be related by theorem.

```
inductive YulExpr
  | lit (n : Nat)
  | hex (n : Nat)
  | str (s : String)
  | ident (name : String)
  | call (func : String) (args : List YulExpr)

inductive YulStmt
  | comment (text : String)
  | let_ (name : String) (value : YulExpr)
  | letMany (names : List String) (value : YulExpr)
  | assign (name : String) (value : YulExpr)
  | expr (e : YulExpr)
  | leave
  | if_ (cond : YulExpr) (body : List YulStmt)
  | for_ (init : List YulStmt) (cond : YulExpr)
     (post : List YulStmt) (body : List YulStmt)
  | switch (expr : YulExpr)
     (cases : List (Nat × List YulStmt))
     (default : Option (List YulStmt))
  | block (stmts : List YulStmt)
  | funcDef (name : String) (params : List String)
     (rets : List String) (body : List YulStmt)

structure YulObject where
  name : String
  deployCode : List YulStmt
  runtimeCode : List YulStmt
```

Listing 11: Core Yul AST used directly by the IR.

```
inductive IRType
  | uint256
  | address
  | unit

structure IRParam where
  name : String
```

```
  ty : IRType

abbrev IRExpr := Yul.YulExpr
abbrev IRStmt := Yul.YulStmt

structure IRFunction where
  name : String
  selector : Nat
  params : List IRParam
  ret : IRType
  payable : Bool := false
  body : List IRStmt

structure IREntrypoint where
  payable : Bool := false
  body : List IRStmt

structure IRContract where
  name : String
  deploy : List IRStmt
  constructorPayable : Bool := false
  functions : List IRFunction
  fallbackEntrypoint : Option IREntrypoint := none
  receiveEntrypoint : Option IREntrypoint := none
  usesMapping : Bool
  internalFunctions : List IRStmt := []
```

Listing 12: IR types, including the direct aliases to Yul expressions and statements.

## D. Layer 2 Proof Architecture

Layer 2 proves `CompilationModel -> IR`. The compilation function first validates the model and selector list, then lowers the validated core to an `IRContract`. The theorem compares `supportedSourceContractSemantics`, the source evaluator used under the `SupportedSpec` witness, with `interpretIR`.

```
def compile (spec : CompilationModel)
    (selectors : List Nat) : Except String IRContract := do
  let validated <- validateCompileInputs spec selectors
  compileValidatedCore validated

theorem compile_preserves_semantics
    (model : CompilationModel)
    (selectors : List Nat)
    (hSupported : SupportedSpec model selectors)
    (ir : IRContract)
    (tx : IRTransaction)
    (initialWorld : ContractState)
    (htxNormalized : TxContextNormalized tx)
    (hcalldataSizeFits : TxCalldataSizeFitsEvm tx)
    (hcompile : compile model selectors = Except.ok ir) :
    sourceResultMatchesIRResult
      (supportedSourceContractSemantics
        model selectors hSupported tx initialWorld)
      (interpretIR ir tx
        (initialIRStateForTx model tx initialWorld))
```

Listing 13: Layer 2 compilation entry point and preservation theorem.

The `SupportedSpec` witness is deliberately decomposed. Global invariants describe selector and layout facts. Surface predicates close unsupported contract-level features such as constructors, events, externals, fallback, and receive. Function predicates then characterize parameter profiles, return profiles, and the body-level statement, state, call, and effect interfaces.

```
structure SupportedBodyInterface
    (spec : CompilationModel) (fn : FunctionSpec) where
  stmtList : SupportedStmtList
    spec.fields (fn.params.map (fun p ⇒p.name)) fn.body
  core : SupportedBodyCoreInterface fn
  state : SupportedBodyStateInterface fn
  calls : SupportedBodyCallInterface spec fn
  effects : SupportedBodyEffectInterface fn
  noLocalObligations : fn.localObligations = []

structure SupportedFunction
    (spec : CompilationModel) (fn : FunctionSpec) where
  nonInternal : fn.isInternal = false
  nonSpecialEntrypoint : isInteropEntrypointName fn.name = false
  params : SupportedParamProfile fn.params
  returns : SupportedReturnProfile fn
  body : SupportedBodyInterface spec fn

structure SupportedSpecInvariants
    (spec : CompilationModel) (selectors : List Nat) : Prop where
  normalizedFields :
    applySlotAliasRanges spec.fields spec.slotAliasRanges = spec.fields
  noPackedFields : forall field, field ∈ spec.fields ->
    field.packedBits = none
  selectorCount :
    selectors.length = (selectorDispatchedFunctions spec).length
  selectorsDistinct : firstDuplicateSelector selectors = none
  functionNamesNodup : (spec.functions.map (fun f ⇒f.name)).Nodup

structure SupportedSpecSurface (spec : CompilationModel) : Prop where
  noConstructor : spec.constructor = none
  noEvents : spec.events = []
  noErrors : spec.errors = []
  noExternals : spec.externals = []
  noFallback : forall fn, fn ∈ spec.functions -> fn.name != "fallback"
  noReceive : forall fn, fn ∈ spec.functions -> fn.name != "receive"

structure SupportedSpec
    (spec : CompilationModel) (selectors : List Nat) where
  invariants : SupportedSpecInvariants spec selectors
  surface : SupportedSpecSurface spec
  functions : forall fn, fn ∈ spec.functions ->
    SupportedFunction spec fn
```

Listing 14: `SupportedSpec` decomposition for the primary Layer 2 theorem, formatted for print width.

Tier 2 uses the same global invariants and surface predicates but replaces the body state interface with `SupportedBodyInterfaceExceptMappingWrites`. This admits the currently proved singleton mapping and struct write shapes through separate theorems such as `compile_preserves_semantics_except_mapping_writes`.

# E. Layer 3 Proof Architecture

Layer 3 proves `IR -> Yul`. Since IR statements are Yul statements, the proof focuses on the generated runtime envelope: selector dispatch, ABI argument loading, inserted helper functions, loop-freedom for the proved fragment, and equivalence between each IR function body and the corresponding Yul body.

```
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 ∈ contract.functions -> DispatchGuardsSafe fn tx)
   (hNoHasSelector : forall fn,
    fn ∈ contract.functions ->
     yulStmtsNoRef "__has_selector" fn.body)
   (hHasSelectorDead : forall fn,
    fn ∈ contract.functions -> HasSelectorDeadBridge fn.body)
   (hLoopFree : forall fn,
    fn ∈ contract.functions -> yulStmtsLoopFree fn.body = true)
   (hbody : forall fn, fn ∈ contract.functions ->
    resultsMatch
     (execIRFunction fn tx.args
        (initialState.withTx tx))
     (interpretYulBody fn tx
        (initialState.withTx tx))) :
   resultsMatch
    (interpretIR contract tx initialState)
    (interpretYulFromIR contract tx initialState)
```

Listing 15: Layer 3 preservation theorem, formatted for print width.

The hypotheses fall into four groups. The selector and calldata bounds (`hselector`, `hNoWrap`) rule out selector and ABI-size overflow cases. The structural hypotheses (`ContractWF`, no fallback, no receive) match the current supported runtime shape. The dispatch hypotheses (`DispatchGuardsSafe`, `yulStmtsNoRef "__has_selector"`, `HasSelectorDeadBridge`) ensure the selector bridge is dead or safe for the generated functions. The body hypotheses require loop-free function bodies and a per-function `resultsMatch` proof, discharged in the compiler by `ir_function_body_equiv` plus a fuel-adequacy lemma.

## E.1. Transaction-State Helper

The theorem listings in Sections 6.1 and E use `initialState.withTx tx` to inject transaction context into an `IRState`:

```
@[reducible] def IRState.withTx
   (s : IRState) (tx : IRTransaction) : IRState :=
  { s with
    sender     := tx.sender
    msgValue   := tx.msgValue
    thisAddress := tx.thisAddress
    blockTimestamp := tx.blockTimestamp
    blockNumber := tx.blockNumber
    chainId    := tx.chainId
```

```
    blobBaseFee := tx.blobBaseFee
    calldata    := tx.args
    selector    := tx.functionSelector }
```

Listing 16: The `IRState.withTx` helper. Every field overwritten by the transaction is listed; all other fields (notably `storage` and `events`) are inherited unchanged.

The `@[reducible]` attribute ensures that the definition unfolds transparently, so existing proofs are unaffected.

## E.2. Dispatch Bridge Hypotheses

The Layer 3 preservation theorem requires two per-contract dispatch facts that are discharged at proof time:

**`HasSelectorDeadBridge`.** Guarantees that generated selector bridge code is semantically dead for the function body under analysis.

**`DispatchGuardsSafe`.** Guarantees that the dispatch guard for each function correctly matches the transaction's selector.

These hypotheses replace the former `keccak256_first_4_bytes` axiom. They are discharged per contract using the kernel-computable Keccak engine, which evaluates selector computations definitionally.