PREPRINT — NOT PEER REVIEWED

Submitted: March 31, 2026

VSEL: Verifiable Semantic Execution Layer

Closing the Gap Between Verified Execution and Correct Execution Through Formally Bound Semantic Proof Systems

Mayckon Giovani

mayckongiovani.xyz

Preprint Notice. This manuscript has not undergone peer review. It is made available to facilitate early dissemination of research findings. The authors welcome feedback and correspondence regarding the technical content. Comments may be directed to the author's contact above.

Subject Areas: Cryptography and Security (cs.CR); Programming Languages (cs.PL); Logic in Computer Science (cs.LO); Formal Languages and Automata Theory (cs.FL)

MSC Classes: 68Q60, 94A60, 03B70, 68Q45

ACM Classes: D.2.4, F.3.1, E.3, C.2.4

Abstract

Contemporary cryptographic execution systems—particularly those employing zero-knowledge proofs—provide strong guarantees that a computation satisfies a given arithmetic circuit. However, satisfying a circuit is not equivalent to executing correctly with respect to the intended semantics of the system being proven. This paper identifies and formalizes the *semantic gap*: the class of failures in which execution is provably valid under a proof system yet provably invalid under the system's formal specification. We present the *Verifiable Semantic Execution Layer* (VSEL), a layered architecture that binds formal specification, execution, constraint derivation, proof generation, and verification into a single semantically coherent pipeline. VSEL models systems as deterministic labeled transition systems, defines explicit semantic mappings between concrete and formal artifacts, derives constraints mechanically from a semantic intermediate representation, and requires that every accepted proof attest not merely to constraint satisfaction but to membership in the formal language of valid execution traces. We define the proof obligations, invariant system, and refinement chain required for end-to-end semantic correctness; characterize the adversarial model including specification manipulation, underconstraint exploitation, and compositional failure; and establish the conditions under which composition of independently correct systems preserves global correctness. The architecture integrates hybrid post-quantum cryptography to ensure long-term validity of proofs and commitments, and introduces a formal economic invariant layer that elevates economic semantics from informal domain knowledge to enforceable first-class predicates over states and execution traces. We provide a complete formal treatment of the system model, semantic preservation theorems, constraint soundness and completeness conditions, witness uniqueness requirements, economic admissibility conditions, and the assume-guarantee framework for safe composition.

Keywords: zero-knowledge proofs, formal verification, semantic correctness, deterministic state machines, constraint derivation, post-quantum cryptography, compositional verification, refinement, invariants, economic invariants, witness uniqueness

Contents

The proliferation of zero-knowledge proof systems in distributed computing—from rollups and verifiable computation platforms to privacy-preserving protocols—has established a widely accepted correctness criterion: an execution is deemed correct if a verifier accepts a proof that the execution satisfies a set of arithmetic constraints. This criterion, while cryptographically sound, is semantically insufficient.

The distinction is precise. *Verified execution* means that a computation satisfies a circuit, i.e., that there exists a witness assignment making all constraint polynomials evaluate to zero over the specified field. *Correct execution* means that the computation belongs to the set of valid behaviors defined by the system's formal specification—that every state transition respects the intended semantics, preserves all required invariants, and produces observables consistent with the system's design.

These two notions coincide only when the circuit is a faithful, complete, and sound encoding of the formal semantics. In practice, this condition is rarely established with rigor. Circuits are written by engineers interpreting informal specifications. The transformation from intended behavior to arithmetic constraints passes through multiple layers of human judgment, each introducing opportunities for semantic drift: the gradual, often invisible divergence between what the system is supposed to do and what the proof system actually verifies.

The consequences are not theoretical. Underconstrained circuits have led to exploitable vulnerabilities in deployed systems, where proofs validate over executions that violate the protocol's intended invariants. The root cause is structural: existing architectures treat the proof system as the ground truth of correctness, when it should be treated as a *projection* of a formally defined ground truth onto a constraint domain.

This paper presents the Verifiable Semantic Execution Layer (VSEL), an architecture that inverts this relationship. In VSEL, correctness is defined by a formal specification expressed as a deterministic labeled transition system. All downstream artifacts—the execution engine, the constraint system, the proof, and the verification procedure—are derived from and bound to this specification through explicit, formally justified refinement relations. The central guarantee is:

$$\text{Verify}(\pi) \implies \text{ValidTrace}(\tau) \tag{1}$$

where $\text{ValidTrace}(\tau)$ denotes membership of the execution trace τ in the language of valid traces induced by the formal specification, not merely satisfaction of an independently written circuit.

The contributions of this paper are as follows. First, we formalize the semantic gap as a precise failure class and characterize the adversarial models that exploit it (Sections 1 and 12). Second, we define the VSEL system model as a deterministic labeled transition system with explicit state structure, transition classes, and observables (Section 2). Third, we introduce the semantic mapping layer that binds concrete execution artifacts to formal semantics through commutativity obligations (Section 3). Fourth, we define the invariant system governing local, global, and temporal correctness (Section 4). Fifth, we specify the execution trace model with commitment structure and sufficiency conditions (Section 5). Sixth, we formalize the constraint derivation pipeline with soundness, completeness, and underconstraint analysis (Section 6). Seventh, we define the proof and verification layers with witness uniqueness and semantic binding requirements (Sections 7 and 8). Eighth, we establish the composition model using assume-guarantee reasoning (Section 10). Ninth, we specify the cryptographic model including hybrid post-quantum considerations (Section 11). Finally, we present a comprehensive adversarial analysis (Section 12) and discuss implementation considerations, limitations, and related work (Sections 13 to 15).

1 Problem Statement

We formalize the gap between verified execution and correct execution.

Definition 1.1 (Execution Correctness)**.** Let $\mathcal{M} = (S, I, T, \Sigma, O)$ be a formal system specification (defined precisely in Section 2). An execution trace $\tau = (s_0, \sigma_0, s_1, \sigma_1, \ldots, s_n)$ is *correct* if and only if:

$$s_0 \in I \quad \wedge \quad \forall\, i \in \{0, \ldots, n{-}1\} : \ (s_i, \sigma_i, s_{i+1}) \in T \tag{2}$$

We write ValidTrace(τ) to denote this predicate.

Definition 1.2 (Verified Execution)**.** Let $\mathcal{C}$ be a constraint system and π a proof. An execution is *verified* if:

$$\text{Verify}(\pi, \mathit{Pub}) = \mathit{true} \tag{3}$$

where Pub denotes the public inputs to the verification procedure.

The implicit assumption in current systems is:

$$\text{Verify}(\pi) = \mathit{true} \implies \text{ValidTrace}(\tau) \tag{4}$$

This implication holds if and only if the constraint system $\mathcal{C}$ is a sound and complete encoding of the formal specification $\mathcal{M}$. Formally, this requires two properties.

Definition 1.3 (Constraint Soundness)**.** SatisfiesConstraints(τ) $\implies$ ValidTrace(τ). No invalid execution satisfies the constraints.

Definition 1.4 (Constraint Completeness)**.** ValidTrace(τ) $\implies$ SatisfiesConstraints(τ). Every valid execution is representable in the constraint system.

When soundness fails, the system accepts proofs for invalid executions—the most dangerous failure class. When completeness fails, valid executions cannot be proven, which is a liveness failure. The semantic gap arises when neither property is formally established, which is the default state of most deployed systems.

The gap is not merely a quality assurance problem. It is a structural consequence of architectures that treat the circuit as the specification rather than as a derived artifact. In such architectures, the question "is this circuit correct?" has no formal referent against which to evaluate it. VSEL eliminates this gap by establishing the formal specification as the sole authority on correctness and requiring that every downstream artifact—including the constraint system—be derived from it through verifiable transformations.

2 System Model

Definition 2.1 (VSEL System)**.** A VSEL system is a deterministic labeled transition system:

$$\mathcal{M} = (S, I, T, \Sigma, O) \tag{5}$$

where:

- S is the set of valid states,
- $I \subseteq S$ is the set of initial states,
- Σ is the set of inputs,
- $T \subseteq S \times \Sigma \times S$ is the transition relation,
- O is the set of observable outputs.

Definition 2.2 (State Structure)**.** A state $s \in S$ is a tuple:

$$s = (C, D, E, \Omega, \tau) \tag{6}$$

where:

- C is the *canonical state*: the minimal, semantically sufficient representation (accounts, storage, system data),
- D is the *derived state*: values functionally determined by C (Merkle roots, aggregates),
- E is the *environment*: external context (timestamps, block height, execution domain),
- Ω is the *economic context*: price oracles, exposure limits, liquidity thresholds, fee schedules, collateral requirements,
- τ is the *trace metadata*: sequence index, prior state commitment, execution epoch.

The derived state and economic context satisfy strict functional dependencies:

Axiom 2.1 (Derived State Determinism)**.**

$$\forall\, s = (C, D, E, \Omega, \tau) \in S :\ D = \text{Derive}(C) \tag{7}$$

where Derive : $\mathcal{C} \to \mathcal{D}$ is a total, deterministic function. Derived state introduces no new semantics.

Axiom 2.2 (Economic Context Determinism)**.**

$$\forall\, s = (C, D, E, \Omega, \tau) \in S :\ \Omega = \text{DeriveEconomic}(C, E) \tag{8}$$

where DeriveEconomic is a total, deterministic function. The economic context is not auxiliary data; it is a formal state component subject to the same invariant, constraint, and proof obligations as all other components.

Definition 2.3 (State Validity)**.** A state is valid if:

$$\text{ValidState}(s) \ \equiv\ P_C(C) \ \wedge\ P_D(D) \ \wedge\ P_E(E) \ \wedge\ P_\tau(\tau) \tag{9}$$

where each P_* is a decidable predicate defining structural and semantic correctness for its component.

Definition 2.4 (Transition Function)**.** The transition function Apply : $S \times \Sigma \to S$ is deterministic:

$$\forall\, s \in S,\ \sigma \in \Sigma :\ \exists!\ s' \text{ such that } \text{Apply}(s, \sigma) = s' \tag{10}$$

No nondeterminism is permitted. If randomness exists in the system, it must be explicitly modeled as a component of E.

Axiom 2.3 (Closure)**.** The state space is closed under transitions:

$$\forall\, s \in S,\ \sigma \in \Sigma :\ \text{Apply}(s, \sigma) \in S \tag{11}$$

No transition produces an invalid state.

Definition 2.5 (Input Structure)**.** An input $\sigma \in \Sigma$ is decomposed as:

$$\sigma = (\sigma_{payload},\ \sigma_{auth},\ \sigma_{aux}) \tag{12}$$

where $\sigma_{payload}$ defines the semantic command, σ_{auth} provides authorization evidence, and σ_{aux} carries auxiliary data (proving hints, optimization flags) that must not influence semantic outcome.

Axiom 2.4 (Auxiliary Data Independence)**.**

$$\forall\, s,\ \sigma_p,\ \sigma_a,\ aux_1,\ aux_2 :\ \text{Apply}(s, (\sigma_p, \sigma_a, aux_1)) = \text{Apply}(s, (\sigma_p, \sigma_a, aux_2)) \tag{13}$$

Definition 2.6 (Observable Function)**.** Observables map transitions to externally visible outputs:

$$\text{Obs} : S \times \Sigma \times S \to O \tag{14}$$

Observables must be derivable from state and input; no hidden side effects are permitted.

2.1 Transition Classes

VSEL partitions the input-state space into disjoint transition classes, each with explicit guard conditions. Let $\mathcal{K} = \{K_{init}, K_{update}, K_{noop}, K_{error}, K_{batch}, K_{reject}\}$ denote the set of transition classes, with associated guard predicates $G_K : S \times \Sigma \to \{true, false\}$.

Definition 2.7 (Transition Partitioning)**.** The guard system satisfies:

$$\text{(Exhaustiveness)} \quad \forall\, s \in S,\ \sigma \in \Sigma : \bigvee_{K \in \mathcal{K}} G_K(s, \sigma) \tag{15}$$

$$\text{(Disjointness)} \quad \forall\, K_1 \neq K_2,\ \forall\, s, \sigma :\ \neg(G_{K_1}(s, \sigma) \wedge G_{K_2}(s, \sigma)) \tag{16}$$

after application of a strict priority ordering. Exhaustiveness ensures every input-state pair is handled. Disjointness ensures the response is unique.

The transition classes are: *initialization* (creating a valid initial state from no prior state), *state update* (standard semantic mutation under valid input, authorization, and preconditions), *no-op* (rejection of invalid or unauthorized inputs with state unchanged), *error* (explicit transition to a defined error state preserving invariants), *batch* (sequential application of multiple inputs, semantically equivalent to iterated single application), and *reject* (catch-all for structurally malformed inputs).

2.2 Execution Pipeline

Each transition follows a strict sequential pipeline:

1. **Input canonicalization**: $\sigma \mapsto \text{Canonical}(\sigma)$. Reject if ambiguous or malformed.
2. **Authorization check**: $\text{Auth}(\sigma) = true$. Reject otherwise.
3. **Precondition validation**: $\text{Pre}(s, \sigma) = true$.
4. **State transformation**: $s' = \text{Apply}(s, \sigma)$.
5. **Postcondition validation**: $\text{Post}(s, \sigma, s') = true$.
6. **Derived state recomputation**: $D' = \text{Derive}(C')$.
7. **Metadata update**: $\tau' = \text{Update}(\tau, s')$.

Deviation from this pipeline at any step constitutes a system failure. The pipeline is not an implementation suggestion; it is a semantic requirement.

3 Semantic Model

The semantic model defines the binding between concrete execution artifacts and the formal specification. This binding is the mechanism by which VSEL ensures that proofs attest to semantic validity, not merely computational consistency.

Definition 3.1 (Semantic Mapping)**.** Let S_c, Σ_c, Tr_c denote the concrete state, input, and trace spaces, and S_f, Σ_f, Tr_f the formal counterparts. The semantic mapping consists of:

$$\mu_S : S_c \to S_f \tag{17}$$

$$\mu_\Sigma : \Sigma_c \to \Sigma_f \tag{18}$$

$$\mu_T : (S_c \times \Sigma_c \times S_c) \to (S_f \times \Sigma_f \times S_f) \tag{19}$$

$$\mu_{Tr} : Tr_c \to Tr_f \tag{20}$$

$$\mu_O : O_c \to O_f \tag{21}$$

These mappings must satisfy the following principles.

Axiom 3.1 (Totality)**.** For every concrete artifact accepted by the execution pipeline, the mapping is defined:

$$\forall\, x \in \mathit{AcceptedConcreteArtifacts} :\ \mu(x) \text{ is defined} \tag{22}$$

Axiom 3.2 (Determinism)**.** Each mapping is a function, not a relation:

$$\forall\, x :\ \mu(x) = y \implies \nexists\, y' \neq y \text{ such that } \mu(x) = y' \tag{23}$$

Axiom 3.3 (Closed Semantic Domain)**.** If a concrete behavior cannot be mapped into the formal model, it is invalid:

$$\forall\, x \notin \mathrm{dom}(\mu) :\ x \text{ is rejected} \tag{24}$$

The central semantic obligation is the commutativity of execution and mapping.

Theorem 3.1 (Execution-Mapping Commutativity)**.** For all admissible concrete states s_c and inputs σ_c:

$$\mu_S(\mathrm{Apply}_c(s_c, \sigma_c)) = \mathrm{Apply}_f(\mu_S(s_c),\ \mu_\Sigma(\sigma_c)) \tag{25}$$

This theorem states that the following diagram commutes:

$$\begin{array}{ccc} s_c & \xrightarrow{\mathrm{Apply}_c} & s_c' \\ {\scriptstyle \mu_S}\downarrow & & \downarrow{\scriptstyle \mu_S} \\ s_f & \xrightarrow[\mathrm{Apply}_f]{} & s_f' \end{array}$$

If this diagram does not commute for any admissible (s_c, σ_c), the concrete execution diverges from the formal model, and any proof over the concrete execution validates the wrong semantics.

Theorem 3.2 (Observable Commutativity)**.** For all admissible transitions (s_c, σ_c, s_c'):

$$\mu_O(\mathrm{Obs}_c(s_c, \sigma_c, s_c')) = \mathrm{Obs}_f(\mu_S(s_c),\ \mu_\Sigma(\sigma_c),\ \mu_S(s_c')) \tag{26}$$

Additional semantic preservation requirements include canonicalization neutrality ($\mu_\Sigma(\sigma) = \mu_\Sigma(\mathrm{Canonical}(\sigma))$ for semantically equivalent inputs), derived state commutativity ($\mu_D(\mathrm{Derive}_c(C_c)) = \mathrm{Derive}_f(\mu_C(C_c))$), and auxiliary data exclusion (Axiom 2.4 lifted to the mapping level).

Definition 3.2 (Semantic Admissibility)**.** A concrete artifact x is semantically admissible if and only if:

$$\mathrm{Canonical}(x) \text{ is defined } \wedge\ \mu(x) \text{ is defined } \wedge\ \mu(x) \text{ satisfies the formal model} \tag{27}$$

Proposition 3.3 (Mapping Soundness)**.** If a concrete artifact is accepted, its formal image is valid:

$$\mathit{Accepted}(x) \implies \mathit{ValidFormal}(\mu(x)) \tag{28}$$

Proposition 3.4 (Mapping Completeness)**.** If a formal execution is intended to be realizable, there exists an admissible concrete realization:

$$\mathit{Realizable}(y) \implies \exists\, x :\ \mu(x) = y \tag{29}$$

Failure of soundness allows invalid executions to pass through the system. Failure of completeness means the formal model describes behaviors the system cannot realize. Both are failures of the semantic layer.

4 Invariants and Correctness Conditions

Invariants define the true correctness surface of the system. If an invariant can be violated, the system is incorrect regardless of proof validity.

Definition 4.1 (Local Invariant)**.** A local invariant L is a predicate over individual transitions:

$$\forall\,(s,\sigma,s') \in T:\ L(s,\sigma,s') \tag{30}$$

The local invariant family includes:
Transition validity (L_{valid}): $\text{Apply}(s,\sigma) = s'$, ensuring no undefined transitions.
State preservation (L_{state}): $\text{ValidState}(s) \wedge \text{ValidState}(s')$, ensuring no transition produces an invalid state.
Resource conservation (L_{cons}): $\text{Total}(C_s) = \text{Total}(C_{s'}) + \Delta_{fees}(\sigma)$, where Total computes the aggregate conserved quantity and Δ_{fees} is the explicitly modeled fee extraction. No hidden creation or destruction of resources is permitted.
Bounded mutation ($L_{bounded}$): $|\text{Diff}(s,s')| \leq \delta_{\max}$, preventing unbounded state changes.
Determinism (L_{det}): $\exists!\, s'$ for each (s,σ), restating Definition 2.4 as an invariant.

Definition 4.2 (Global Invariant)**.** A global invariant G is a predicate over every reachable state:

$$\forall\, s \in \text{Reachable}(I,T):\ G(s) \tag{31}$$

Global invariants include state validity (G_{valid}), structural integrity (G_{struct}: consistent encoding and well-formedness), commitment consistency (G_{commit}: $\text{Commit}(C) = D.commitment$), monotonic metadata ($G_{mono}$: $\tau_{current} \geq \tau_{previous}$), and environment consistency (G_{env}).

Definition 4.3 (Temporal Invariant)**.** A temporal invariant $TInv$ is a predicate over complete execution traces:

$$\forall\,\tau \in Traces(\mathcal{M}):\ TInv(\tau) \tag{32}$$

Temporal invariants include trace validity (T_{valid}: all transitions in the trace are valid), no state reversion (T_{no_revert}: no rollback of committed state), cumulative resource consistency (T_{cons}: conservation holds over the entire trace), causality preservation (T_{causal}: input ordering implies state ordering), and trace completeness ($T_{complete}$: every state change is recorded).

Lemma 4.1 (Inductive Invariance)**.** If $G(s_0)$ holds for all $s_0 \in I$, and $G(s) \implies G(\text{Apply}(s,\sigma))$ for all $s \in S$ and $\sigma \in \Sigma$, then G holds for all reachable states:

$$\forall\,\tau = (s_0,\ldots,s_n)\ \text{valid}:\ \forall\, i:\ G(s_i) \tag{33}$$

Proof. By induction on trace length. The base case holds by the initial state condition. The inductive step follows from the preservation condition applied to each transition. □

The invariant system is complete if no semantically invalid execution satisfies all invariants. Formally:

Definition 4.4 (Invariant Completeness)**.** The invariant set $\{L_i\} \cup \{G_j\} \cup \{TInv_k\}$ is complete if:

$$\nexists\,\tau:\ \left(\bigwedge_i L_i(\tau)\ \wedge\ \bigwedge_j G_j(\tau)\ \wedge\ \bigwedge_k TInv_k(\tau)\right)\ \wedge\ \neg\text{ValidTrace}(\tau) \tag{34}$$

If such a trace exists, the invariant set is insufficient and the system is vulnerable to executions that satisfy all defined properties yet are semantically invalid. This is the most dangerous class of invariant failure because it is invisible to all defined checks.

4.1 Economic Invariants

VSEL treats economic semantics as a first-class invariant domain, orthogonal to structural invariants but equally mandatory. The state validity predicate is extended:

Definition 4.5 (Economic Validity)**.**

$$\text{EconomicallyValid}(s) \equiv P_\Omega(C, \Omega) \wedge \text{WithinLimits}(\Omega) \wedge \text{NoExploitableConfiguration}(s) \quad (35)$$

Definition 4.6 (Admissibility)**.** A state is admissible if and only if it is both structurally valid and economically valid:

$$\text{Admissible}(s) \equiv \text{ValidState}(s) \wedge \text{EconomicallyValid}(s) \quad (36)$$

Neither predicate subsumes the other. Both are required.

Economic invariants are classified into the same four categories as structural invariants.
Local economic invariants constrain individual transitions. E_{cost}: no resource may be acquired at zero cost when the resource has positive value ($\text{ResourceAcquired}(s, s') > 0 \implies \text{Cost}(\sigma) > 0$). $E_{leverage}$: no transition may result in effective leverage exceeding the system-defined maximum ($\text{EffectiveLeverage}(s', entity) \leq \text{MaxLeverage}(\Omega)$). $E_{proportionality}$: fees must be proportional to economic impact. $E_{slippage}$: price impact of any single trade must be bounded relative to pool depth. $E_{collateral}$: every position must maintain sufficient collateral after any transition.
Global economic invariants constrain every reachable state. $G_{solvency}$: total assets must exceed total liabilities ($\text{TotalAssets}(s) \geq \text{TotalLiabilities}(s)$). $G_{concentration}$: no single entity may hold a disproportionate share of any resource. $G_{liquidity}$: liquidity pools must maintain minimum depth. G_{dust}: no account may hold a positive balance below the dust threshold.
Temporal economic invariants constrain execution sequences. These are particularly important because most economic attacks manifest as sequences, not single steps. $TE_{extraction}$: no entity may extract more than a defined threshold within a single epoch:

$$\forall\, \tau_{epoch},\ \forall\, entity :\ \text{NetExtraction}(\tau_{epoch}, entity) \leq \text{MaxExtraction}(\Omega, entity) \quad (37)$$

TE_{flash}: flash operations must either bound uncollateralized exposure at every intermediate state or maintain collateral throughout. $TE_{sandwich}$: no trace may contain a sandwich pattern where one entity profits by surrounding another entity's transaction. $TE_{velocity}$: resource turnover within an epoch is bounded, preventing wash trading.
Compositional economic invariants apply when systems are composed. $CE_{arbitrage}$: cross-system arbitrage profit is bounded. $CE_{contagion}$: economic failure in one system has bounded impact on composed systems.

The proof obligation for economic invariants mirrors the structural case:

$$\forall\, (s, \sigma, s') \in T :\ \text{EconomicallyValid}(s) \implies \text{EconomicallyValid}(s') \quad (38)$$

The end-to-end guarantee is strengthened from $\text{Verify}(\pi) \implies \text{ValidTrace}(\tau)$ to:

$$\text{Verify}(\pi) \implies \text{AdmissibleTrace}(\tau) \quad (39)$$

where $\text{AdmissibleTrace}(\tau) \equiv \text{ValidTrace}(\tau) \wedge \forall\, s_i \in \tau : \text{EconomicallyValid}(s_i) \wedge \forall\, TE_{econ} : TE_{econ}(\tau)$.

Economic invariants are parameterized by the economic context Ω, allowing different deployments to define different economic parameters while sharing the same formal enforcement machinery. The parameter space itself must be constrained: no parameter combination may render the system economically unsound.

The integration of economic invariants into the formal specification follows a defined process: domain experts express economic properties in semi-formal language, formal methods engineers translate them into predicates over states and traces, and the resulting invariants enter the standard pipeline of proof obligations, counterexample construction, constraint encoding, and coverage verification. This process is not optional; it is the mechanism by which economic intent becomes formal specification.

4.2 Cross-Layer Invariants

VSEL enforces consistency across its architectural layers through cross-layer invariants:
Execution–Specification (X_{exec}): $\text{Apply}_{impl}(s, \sigma) = \text{Apply}_{spec}(s, \sigma)$ for all admissible (s, σ).
Specification–Constraints ($X_{constraint}$): $\text{ValidTrace}(\tau) \iff \text{SatisfiesConstraints}(\tau)$.
Execution–Proof (X_{proof}): $\text{Verify}(\pi) \implies \text{ValidTrace}(\tau)$.

These invariants are not independently enforceable; they are consequences of the refinement chain described in Section 9.

5 Execution Trace Model

Definition 5.1 (Execution Trace)**.** An execution trace is a sequence:

$$\tau = (s_0, \sigma_0, s_1, \sigma_1, \ldots, \sigma_{n-1}, s_n) \tag{40}$$

such that $s_0 \in I$ and $(s_i, \sigma_i, s_{i+1}) \in T$ for all i.

Definition 5.2 (Trace Entry)**.** Each transition is recorded as a concrete trace entry:

$$e_i = (id_i,\ s_i,\ \sigma_i,\ s_{i+1},\ o_i,\ meta_i) \tag{41}$$

where $id_i = i$ (strict sequential indexing), $o_i = \text{Obs}(s_i, \sigma_i, s_{i+1})$, and $meta_i$ includes timestamp, execution domain, and prior commitment.

The trace is not a log. Logs are optional, may be incomplete, and carry no semantic obligation. The trace is the canonical record of execution. If a state change is not in the trace, it did not happen within the system's semantic model.

5.1 Commitment Structure

The trace is committed via an incremental hash chain:

$$h_0 = \text{Hash}(\text{Commit}(e_0)), \qquad h_{i+1} = \text{Hash}(h_i \parallel \text{Commit}(e_i)) \tag{42}$$

This structure is tamper-evident, order-preserving, and deterministic. Any modification to any entry changes the final commitment h_n.

Definition 5.3 (Trace Commitment)**.**

$$\text{Commit}(\tau) = h_n \tag{43}$$

5.2 Trace Sufficiency

A trace is *sufficient* if it uniquely determines the semantic execution.

Theorem 5.1 (Trace Determines Execution)**.** Given initial state s_0 and input sequence $(\sigma_0, \ldots, \sigma_{n-1})$, the execution trace is uniquely determined:

$$\text{Reconstruct}(s_0, \sigma_0, \ldots, \sigma_{n-1}) = \tau \tag{44}$$

Proof. Follows directly from the determinism of Apply (Definition 2.4). Each $s_{i+1} = \text{Apply}(s_i, \sigma_i)$ is unique, so the entire state sequence is determined by s_0 and the input sequence. □

Theorem 5.2 (Commitment Determines Trace)**.** Under the collision resistance of Hash:

$$\text{Commit}(\tau_1) = \text{Commit}(\tau_2) \implies \tau_1 = \tau_2 \quad \text{(with overwhelming probability)} \tag{45}$$

Definition 5.4 (Replay Determinism)**.**

$$\text{Replay}(\tau) = \tau \tag{46}$$

Given identical initial state, input sequence, and environment, the system produces an identical trace.

The sufficiency conditions require that the trace records: (1) the full initial state or a commitment from which it can be retrieved, (2) all inputs in canonical form, (3) all observables, (4) deterministic ordering, (5) environment context for each transition, and (6) no hidden transitions—every semantically relevant state change has a corresponding trace entry.

6 Constraint System Derivation

The constraint derivation layer transforms formal semantics into an arithmetic constraint system suitable for proof generation. This transformation is the most failure-prone component of any ZK-based architecture, and VSEL treats it accordingly.

Definition 6.1 (Constraint Derivation)**.** Let *SIR* denote the Semantic Intermediate Representation—a deterministic, fully typed, closed representation derived from the formal specification. The constraint derivation is a function:

$$\mathcal{D} : \mathit{SIR} \to \mathcal{C} \tag{47}$$

where $\mathcal{C}$ is the resulting constraint system. $\mathcal{D}$ must be deterministic: the same SIR program always produces the same constraint system.

The SIR serves as the canonical bridge between the formal specification and the constraint system. Every SIR construct must be traceable to a formal semantic definition, and no SIR construct may introduce behavior not present in the specification.

6.1 Soundness and Completeness

Theorem 6.1 (Constraint Soundness)**.**

$$\forall\, \tau :\ \text{SatisfiesConstraints}(\tau) \implies \text{ValidTrace}(\tau) \tag{48}$$

No invalid execution satisfies the derived constraints.

Theorem 6.2 (Constraint Completeness)**.**

$$\forall\, \tau :\ \text{ValidTrace}(\tau) \implies \text{SatisfiesConstraints}(\tau) \tag{49}$$

Every valid execution is representable in the constraint system.

Together, these establish the biconditional:

$$\text{ValidTrace}(\tau) \iff \text{SatisfiesConstraints}(\tau) \tag{50}$$

6.2 Constraint Domains

Constraints are derived over five domains:
State constraints enforce canonical state validity ($C \in \mathit{ValidC}$), derived state consistency ($D = \text{Derive}(C)$), environment validity ($E \in \mathit{ValidE}$), and metadata consistency ($\tau_{i+1} = \text{Update}(\tau_i)$).
Input constraints enforce canonical form ($\sigma = \text{Canonical}(\sigma)$), authorization ($\text{Auth}(\sigma) = \mathit{true}$), and domain validity ($\sigma \in \Sigma$).
Transition constraints enforce functional correctness ($s' = \text{Apply}(s, \sigma)$), mutation scope ($\text{Diff}(s, s') \subseteq \mathit{AllowedMutations}(\sigma)$), and the absence of implicit state changes.
Invariant constraints encode all local, global, and temporal invariants.
Trace constraints enforce transition linking ($s_{i+1} = \text{Apply}(s_i, \sigma_i)$), ordering, and completeness.

6.3 Underconstraint Analysis

Underconstraint is the condition where the constraint system accepts executions that the formal specification rejects. It is the single most exploitable vulnerability class in ZK-based systems.

Definition 6.2 (Underconstraint Types)**.** We identify the following underconstraint classes:

- **Free variable** (U1): a witness variable referenced by no constraint.
- **Weakly constrained variable** (U2): a variable whose value is not uniquely determined by public inputs and other constrained values.
- **Missing branch constraint** (U3): a conditional path in SIR that generates no constraints for one or more branches.
- **Structural-only constraint** (U4): a constraint enforcing format but not semantic meaning.
- **Carry-over omission** (U5): a non-mutated state field lacking an explicit equality constraint to its prior value.
- **Temporal gap** (U6): per-step constraints that fail to enforce cross-step properties.

The carry-over omission (U5) deserves emphasis. In most constraint systems, variables not mentioned in any constraint are *free*, not implicitly carried over. If a state field should remain unchanged during a transition but lacks an explicit equality constraint $s'.f = s.f$, an adversary can set that field to any value. This is the most commonly missing constraint family in deployed ZK systems.

VSEL requires a *constraint coverage matrix* relating every semantic property to the specific constraints that enforce it, for every transition class. Every cell in this matrix must be populated. An empty cell is a vulnerability.

6.4 Witness Uniqueness

Definition 6.3 (Witness Semantic Uniqueness)**.** For all witnesses W_1, W_2 satisfying the constraint system with the same public inputs Pub:

$$\text{Semantics}(W_1) = \text{Semantics}(W_2) \tag{51}$$

Multiple witnesses may exist (e.g., different Merkle path representations), but they must all represent the same semantic execution. If two witnesses encode different state sequences but produce the same public outputs, the proof is semantically ambiguous—it says "something happened" without committing to what.

Witness malleability attacks include state substitution (replacing intermediate states), input substitution (replacing the input sequence), observable manipulation (fabricating outputs),

authorization rebinding (using a signature intended for one payload to authorize another), and temporal reordering (changing the execution sequence).

VSEL prevents these through sufficient constraint coverage: every semantic variable in the witness must be uniquely determined by the public inputs and the constraint system, or must belong to a formally justified equivalence class that does not affect semantic interpretation.

7 Proof System Integration

Definition 7.1 (Proof Statement)**.** Given an execution trace τ and constraint system $\mathcal{C}$ derived from the formal specification:

$$\pi = \text{Prove}(\tau, \mathcal{C}) \tag{52}$$

such that:

$$\text{Verify}(\pi) \implies \text{ValidTrace}(\tau) \tag{53}$$

This is the only statement that matters. The proof attests to semantic validity under the formal specification, not merely to satisfaction of an independently written circuit.

Definition 7.2 (Proof Object)**.** A proof object consists of:

$$\pi = (\mathit{Com},\ W,\ \mathit{Aux},\ \mathit{Meta}) \tag{54}$$

where *Com* denotes commitments to trace and state, W is the witness data, *Aux* contains auxiliary proof artifacts (queries, openings), and *Meta* includes domain separation tags, versioning, and parameters.

Definition 7.3 (Public Inputs)**.** The proof exposes:

$$\mathit{Pub} = (\mathit{root}_{\mathit{init}},\ \mathit{root}_{\mathit{final}},\ \mathit{inputs},\ \mathit{outputs},\ \mathit{domain}) \tag{55}$$

where $\mathit{root}_{\mathit{init}} = \text{Commit}(s_0)$, $\mathit{root}_{\mathit{final}} = \text{Commit}(s_n)$, *inputs* and *outputs* are the observable inputs and outputs, and *domain* is the domain separation identifier.

7.1 Binding Requirements

The proof must bind to the *entire* execution trace, not merely to the initial and final states. Partial binding allows arbitrary intermediate behavior.

$$\mathit{root}_{\mathit{init}} = \text{Commit}(s_0), \qquad \mathit{root}_{\mathit{final}} = \text{Commit}(s_n) \tag{56}$$

Additionally, all observables must be included in or derivable from the public inputs:

$$\forall\, o_i \in \text{Obs}(\tau) :\ o_i \subseteq \mathit{Pub} \ \vee\ o_i = f(\mathit{Pub}) \tag{57}$$

7.2 Domain Separation

Every proof includes a domain identifier:

$$\text{Domain} = \text{Hash}(\mathit{context}) \tag{58}$$

where *context* encodes the system identity, version, and execution epoch. This prevents cross-system proof replay and cross-protocol confusion.

7.3 Proof System Choices

VSEL is agnostic to the specific proof system, supporting STARKs (transparent setup, hash-based security, post-quantum friendly), SNARKs (succinct proofs, pairing-based assumptions, optional trusted setup), and hybrid systems (STARKs for base proofs, SNARKs for recursion and compression). The choice affects performance and trust assumptions but must not affect the semantics of what is proven. Regardless of the proof system:

$$\text{Verify}(\pi) \implies \text{ValidTrace}(\tau) \tag{59}$$

7.4 Soundness and Knowledge Soundness

Definition 7.4 (Proof Soundness)**.**

$$\Pr[\text{invalid } \tau \text{ accepted}] \leq \epsilon \tag{60}$$

where ϵ is negligible in the security parameter.

Definition 7.5 (Knowledge Soundness)**.** There exists an extractor E such that for any prover producing a valid proof π:

$$E(\pi) = W \quad \text{where } W \text{ is a valid witness for } \tau \tag{61}$$

Knowledge soundness prevents proof forgery without knowledge of a valid execution.

7.5 Proof Composition

Proofs may be composed:

$$\pi_{combined} = \text{Compose}(\pi_1, \pi_2, \ldots, \pi_n) \tag{62}$$

with requirements for compositional correctness, invariant preservation, and consistent state chaining. Recursive proofs embed verification of inner proofs within outer proof constraints:

$$\text{Verify}(\pi_{inner}) \subseteq \text{Constraints}(\pi_{outer}) \tag{63}$$

8 Verification Model

The verifier is the final authority in VSEL. If it accepts an invalid execution, the system is broken regardless of the rigor of every other component.

Definition 8.1 (Verification)**.** Given proof π and public inputs Pub:

$$\text{Verify}(\pi, Pub) = \mathit{true} \implies \text{ValidTrace}(\tau) \tag{64}$$

where τ is the execution trace implicitly or explicitly represented by the proof.

The verification procedure is strictly defined as a sequential pipeline:
Step 1: Domain validation. $\text{Domain}(Pub) = \mathit{ExpectedDomain}(\mathit{Context})$. Prevents cross-system replay.
Step 2: Structural validation. Proof format, encoding correctness, parameter consistency. Malformed proofs are rejected immediately.
Step 3: Commitment validation. $root_{init}, root_{final} \in \mathit{ValidCommitments}$. Optionally verified against known state.
Step 4: Cryptographic verification. $\text{Verify}_{crypto}(\pi, Pub) = \mathit{true}$. Polynomial commitment checks, query checks, consistency checks.

Step 5: Semantic binding validation. Public inputs correspond to semantic observables. State commitments correspond to valid states. This step enforces $\pi \implies \text{ValidTrace}(\tau)$, not merely $\pi \implies \text{SatisfiesConstraints}(\tau)$.
Step 6: Invariant enforcement. If invariants are partially externalized, verify invariant commitments and proofs.
Step 7: Acceptance. $\text{Accept}(\pi) \iff$ all checks pass.

VSEL distinguishes between stateless verification (checking proof validity and internal consistency without tracking system state) and stateful verification (maintaining the latest state commitment and trace continuity, checking $root_{prev} = root_{expected}$). Stateful verification provides stronger guarantees by detecting invalid state transitions across proofs.

The most dangerous verification failure mode is *constraint-semantic drift*: the verifier accepts a proof that satisfies constraints but violates the formal specification. This occurs when the constraint system diverges from the specification—precisely the gap VSEL is designed to eliminate through the refinement chain.

9 Refinement Chain

The end-to-end guarantee of VSEL (Equation (1)) is not established by a single proof but by a chain of refinement relations connecting each architectural layer to the one above it.

Definition 9.1 (Refinement Levels)**.** VSEL defines five refinement levels:

$$L_0 : \text{Abstract Specification (formal state machine)} \tag{65}$$
$$L_1 : \text{Semantic Intermediate Representation (SIR)} \tag{66}$$
$$L_2 : \text{Concrete State Machine (implementation)} \tag{67}$$
$$L_3 : \text{Constraint Model} \tag{68}$$
$$L_4 : \text{Proof Statement} \tag{69}$$

Definition 9.2 (Refinement Relation)**.** A refinement relation $R_{i,i+1}$ between levels L_i and L_{i+1} is a mapping $R : \text{States}(L_{i+1}) \to \text{States}(L_i)$ satisfying:
Simulation: Every concrete transition corresponds to an abstract transition:

$$\text{Apply}_{i+1}(s_c, \sigma_c) = s'_c \implies \text{Apply}_i(R(s_c), R_\sigma(\sigma_c)) = R(s'_c) \tag{70}$$

Invariant preservation: Concrete invariants imply abstract invariants through the refinement map:

$$G_{i+1}(s_c) \implies G_i(R(s_c)) \tag{71}$$

Observable preservation: Observables are preserved:

$$\text{Obs}_i(R(s_c), R_\sigma(\sigma_c), R(s'_c)) = R_O(\text{Obs}_{i+1}(s_c, \sigma_c, s'_c)) \tag{72}$$

The composition of all refinements yields the end-to-end guarantee:

$$\text{Verify}(\pi) \xRightarrow{R_{3,4}} \text{SatisfiesConstraints}(\tau) \xRightarrow{R_{2,3}} \textit{ValidConcrete}(\tau_c) \xRightarrow{R_{1,2}} \textit{ValidSIR}(\tau_{sir}) \xRightarrow{R_{0,1}} \text{ValidTrace}(\tau_f) \tag{73}$$

Each implication must be independently justified. If any refinement relation is broken, the end-to-end guarantee does not hold, and the system may produce valid proofs for semantically invalid executions.

The refinement from L_0 to L_1 (abstract specification to SIR) is discharged by theorem proving, establishing that the SIR is a faithful representation of the formal model. The refinement from L_1 to L_2 (SIR to concrete) corresponds to the semantic mapping (Section 3) and is discharged by a combination of theorem proving and differential testing. The refinement from

L_2 to L_3 (concrete to constraints) is the constraint soundness and completeness result (Theorems 6.1 and 6.2), discharged by constraint analysis and adversarial testing. The refinement from L_3 to L_4 (constraints to proof) relies on the cryptographic soundness of the proof system (Definition 7.4).

10 Composition Model

Systems rarely exist in isolation. VSEL defines composition semantics to ensure that correctness survives interaction.

Definition 10.1 (System Composition)**.** Given systems $\mathcal{M}_A = (S_A, I_A, T_A, \Sigma_A, O_A)$ and $\mathcal{M}_B = (S_B, I_B, T_B, \Sigma_B, O_B)$, the composed system is:

$$\mathcal{M}_{A \circ B} = (S_{AB}, I_{AB}, T_{AB}, \Sigma_{AB}, O_{AB}) \tag{74}$$

where $S_{AB} \subseteq S_A \times S_B$, and $T_{AB} \subseteq T_A \cup T_B \cup T_{cross}$ with cross-transitions:

$$T_{cross} \subseteq S_A \times S_B \times \Sigma_{cross} \times S_A \times S_B \tag{75}$$

Composition is not free. The fundamental observation is:

$$\mathit{Correct}(\mathcal{M}_A) \ \wedge \ \mathit{Correct}(\mathcal{M}_B) \ \not\Rightarrow \ \mathit{Correct}(\mathcal{M}_{A \circ B}) \tag{76}$$

Local correctness does not imply global correctness. Composition requires new invariants and new proofs.

10.1 Assume-Guarantee Reasoning

VSEL structures composition through formal contracts. Each subsystem M defines:

$$\text{Contract}(M) = (A(M),\ G(M),\ E(M),\ \mathit{Eff}(M),\ F(M),\ \mathit{Temp}(M)) \tag{77}$$

where $A(M)$ is the set of assumptions M requires from its environment, $G(M)$ is the set of guarantees M provides, $E(M)$ is the observable export interface, $\mathit{Eff}(M)$ is the set of effects visible to other systems, $F(M)$ is the set of forbidden interactions, and $\mathit{Temp}(M)$ is the set of temporal obligations.

Theorem 10.1 (Composition Validity)**.** Composition of $\mathcal{M}_A$ and $\mathcal{M}_B$ is valid if and only if:

$$G(\mathcal{M}_A) \supseteq A(\mathcal{M}_B) \tag{78}$$
$$G(\mathcal{M}_B) \supseteq A(\mathcal{M}_A) \tag{79}$$
$$\mathit{Eff}(\mathcal{M}_A) \cap F(\mathcal{M}_B) = \emptyset \tag{80}$$
$$\mathit{Eff}(\mathcal{M}_B) \cap F(\mathcal{M}_A) = \emptyset \tag{81}$$
$$\mathit{Temp}(\mathcal{M}_A) \wedge \mathit{Temp}(\mathcal{M}_B) \text{ is satisfiable} \tag{82}$$

and no reachable composed state escapes the valid state space:

$$\forall\, s \in \text{Reachable}(\mathcal{M}_{A \circ B}) :\ s \in S_A \times S_B \tag{83}$$

Conditions (78)–(79) ensure mutual assumption satisfaction. Conditions (80)–(81) ensure no forbidden interactions. Condition (82) ensures temporal compatibility. Condition (83) ensures state space closure under composition.

10.2 Cross-System Invariants

Composition introduces invariants that do not exist in either subsystem alone:
Resource conservation: $\text{Total}_A + \text{Total}_B = \textit{constant}$, preventing double-spend across boundaries.
Shared state consistency: $\text{View}_A(\textit{shared}) = \text{View}_B(\textit{shared})$ at every synchronization point.
Causal consistency: if transition t_1 causally precedes t_2, both systems observe this ordering.
Authorization consistency: cross-system inputs must be independently authorized by both systems.

Proof composition combines individual proofs with a cross-invariant proof:

$$\pi_{AB} = \text{Combine}(\pi_A, \pi_B, \pi_{cross}) \tag{84}$$

where π_{cross} attests to the satisfaction of all cross-system invariants.

11 Cryptographic Model

VSEL operates under a hybrid adversarial model encompassing both classical and quantum-capable adversaries. The cryptographic layer does not make the system correct; it makes it hard to cheat. The semantic layer (Sections 3 and 4) defines what correctness means; the cryptographic layer ensures that violations are computationally infeasible.

11.1 Primitives

Hash functions are used for state commitments, Merkle structures, and domain separation. Requirements: collision resistance, preimage resistance, and quantum resistance for long-term artifacts. Candidates include SHA-3/Keccak, BLAKE3, and STARK-friendly hashes (Poseidon, Rescue) for proof-internal use.
Commitment schemes are used for state, trace, and polynomial commitments. Properties: binding and optionally hiding. Implementations include Merkle commitments and polynomial commitments (FRI for STARKs, KZG for SNARKs).
Digital signatures authorize inputs. Requirements: unforgeability under adaptive chosen-message attacks and post-quantum security for long-term validity. VSEL adopts hybrid signatures:

$$\textit{Sig} = (\textit{Sig}_{\textit{classical}},\ \textit{Sig}_{\textit{PQC}}) \tag{85}$$

where both components must verify. Classical candidates include ECDSA and Ed25519; post-quantum candidates include ML-DSA (Dilithium) and Falcon.
Key exchange uses hybrid schemes combining ECDH with ML-KEM (Kyber) to ensure both current performance and future security.

11.2 Domain Separation

All cryptographic operations include domain separation:

$$\text{Hash}(\textit{domain} \parallel \textit{data}) \tag{86}$$

preventing cross-protocol attacks and replay across contexts. The domain identifier encodes system identity, version, and execution epoch.

11.3 Post-Quantum Considerations

The threat model includes "harvest now, decrypt later" attacks, where an adversary records current artifacts and breaks them when quantum computers become available. VSEL classifies every cryptographic artifact by temporal sensitivity:
Ephemeral (seconds to minutes): session keys, temporary commitments. Classical primitives suffice.
Session (hours to days): execution-batch commitments. Hybrid recommended.
Archival (months to years): state commitments, proofs, audit evidence. Hybrid required.
Permanent (decades): historical trace commitments, compliance evidence. Quantum-resistant primitives mandatory.

For proof systems, STARKs provide natural post-quantum resistance through hash-based security. SNARKs based on pairing assumptions are quantum-vulnerable and are used only for compression, with the understanding that re-proving under a quantum-resistant system is possible if witness data is archived.

Migration protocols are defined for commitment migration (re-hashing under a new function with attestation), signature migration (re-signing with attestation chain), and proof migration (re-proving from archived witnesses). The critical requirement is that witness data must be archived for the lifetime of the proof's relevance; without it, re-proving is impossible.

11.4 Long-Term Validity

The hybrid strategy ensures that if either the classical or post-quantum component remains secure, the system remains secure:

$$Security = Classical \ \wedge \ PostQuantum \tag{87}$$

Proofs remain valid as long as the underlying cryptographic assumptions hold. When assumptions are threatened, the migration protocols enable transition to new primitives without invalidating the semantic guarantees established by the formal specification.

12 Adversarial Analysis

VSEL assumes a deliberately uncomfortable adversarial posture: the primary source of failure is not execution or proof, but the definition of correctness itself. If correctness is underspecified, the system can be perfectly verified and completely wrong.

12.1 Adversary Classes

Malicious prover. Controls proof generation. Objectives: produce valid proofs for invalid executions by exploiting underconstrained circuits or manipulating witness data. Capabilities: full control over witness inputs, knowledge of the constraint system, ability to search constraint edge cases.
Malicious executor. Controls the execution environment. Objectives: produce state transitions outside the formal model, introduce invalid states, diverge from specified semantics. Capabilities: control over runtime behavior, manipulation of execution ordering and timing.
Specification manipulator. Targets the system through its specification. Objectives: introduce ambiguity, omit edge cases, define incomplete invariants. This adversary is frequently internal and consistently underestimated.
Constraint-level attacker. Targets the constraint system directly. Objectives: find executions that satisfy constraints but violate semantics. Capabilities: circuit analysis, underconstraint discovery, witness manipulation.

Economic adversary. Rational, profit-driven. Objectives: exploit inconsistencies for financial gain, manipulate cross-system composition. Capabilities: multi-system interaction, timing exploitation, arbitrage of semantic gaps.

12.2 Attack Surfaces

Semantic drift. The most fundamental attack surface. Mismatch between the formal specification, the implementation, and the constraint system. The adversary does not need to break cryptography; they only need to find where the system's definition is incomplete or ambiguous. VSEL defends against this through the refinement chain (Section 9) and the semantic mapping commutativity obligation (Theorem 3.1).

Underconstrained systems. Constraints fail to fully encode semantics, allowing multiple valid witnesses for semantically invalid behavior. VSEL defends through systematic underconstraint analysis (Section 6.3), the constraint coverage matrix, and the invalid execution witness suite—a systematic construction of witness families that the constraint system must reject.

Trace incompleteness. Incomplete capture of execution history allows validation of partial behavior only. VSEL defends through trace sufficiency conditions (Section 5.2) and the commitment chain (Section 5.1).

Proof-valid-but-wrong execution. A proof validates over an execution that satisfies constraints but violates the formal specification. This is the canonical manifestation of the semantic gap. VSEL eliminates this through the biconditional (50): if constraints are sound and complete with respect to the specification, no such execution exists.

Composition failures. Locally correct systems produce globally invalid behavior. Each system preserves its own invariants, but cross-system invariants are violated. VSEL defends through assume-guarantee reasoning (Section 10.1) and explicit cross-invariant enforcement.

Temporal attacks. Exploitation across time rather than within a single execution. Invariants hold per step but fail over sequences due to accumulation (rounding errors, counter overflow, resource drift). VSEL defends through temporal invariants (Definition 4.3) and long-trace simulation.

12.3 Concrete Attack Scenarios

Scenario 1: Valid proof, invalid semantics. An adversary discovers that the constraint system does not enforce resource conservation for a specific transition class. They construct a witness where resources are created from nothing, generate a valid proof, and submit it. The verifier accepts because the proof satisfies all constraints. The system loses funds. Root cause: constraint incompleteness (violation of Theorem 6.1).

Scenario 2: Witness ambiguity exploitation. The constraint system allows two distinct witness assignments for the same public inputs, representing different state transitions. An adversary uses this to claim one execution occurred while actually performing another. Root cause: witness semantic non-uniqueness (violation of Definition 6.3).

Scenario 3: Compositional double-spend. Two systems share a resource pool. A cross-system transfer debits in system A but the credit in system B is not atomically linked. The adversary exploits the timing gap to spend the resource in both systems. Root cause: missing cross-system conservation invariant (violation of Section 10.2).

Scenario 4: Specification blind spot. The formal specification does not define behavior for a particular input class. The implementation handles it with a default that happens to preserve all defined invariants but violates the intended economic semantics. VSEL mitigates this through the economic invariant layer (Section 4.1), which formalizes economic intent as enforceable predicates. However, the risk of specification incompleteness remains for economic properties not yet identified by domain experts.

Scenario 5: Flash loan extraction. An adversary executes an atomic sequence—borrow, manipulate price, trade at manipulated price, repay—where each transition individually preserves all structural invariants and resource conservation. The adversary profits; liquidity providers lose. Root cause: absence of temporal economic invariants. VSEL defends through $TE_{extraction}$ (bounded epoch extraction) and TE_{flash} (flash operation collateral requirements at every intermediate state).
Scenario 6: Sandwich attack. An adversary front-runs a victim's transaction, causing the victim to execute at a worse price, then back-runs to capture the profit. Each transaction is individually valid; conservation holds. VSEL defends through $TE_{sandwich}$, which formalizes the prohibition of profit extraction through transaction ordering manipulation.

12.4 Defense Summary

VSEL's defense is layered: semantic closure (all behavior must be specified), constraint completeness (all semantics must be encoded), trace-level verification (validation over complete execution histories), cross-layer consistency (refinement chain), and adversarial testing (systematic construction of counterexamples and invalid witnesses). The residual risk is specification incompleteness—the possibility that the formal model itself fails to capture the intended system behavior. This risk cannot be eliminated by any formal method; it can only be bounded through rigorous specification practices, adversarial self-audit, and continuous validation.

13 Implementation Considerations

The translation from VSEL's formal architecture to a deployable system requires mapping each theoretical component to concrete engineering artifacts. This section specifies the implementation requirements without prescribing specific technologies.
Formal Specification Engine. The abstract specification (Section 2) must be expressed in a machine-checkable language. TLA+ is suitable for state machine semantics and model checking of bounded instances. Coq, Lean 4, or Isabelle/HOL are required for theorem-level guarantees, particularly for the refinement proofs (Section 9) and universal invariant preservation (Theorem 4.1). The specification is not documentation; it is a formal artifact from which downstream components are derived.
Execution Engine. The execution engine implements the transition function Apply with strict adherence to the pipeline (Section 2.2). Determinism must be enforced at the implementation level: no floating-point arithmetic, no implicit ordering dependencies, no hidden randomness. The engine must emit sufficient state and trace material for semantic reconstruction, as required by the trace sufficiency conditions (Section 5.2).
Trace Engine. Every transition must be recorded as a trace entry (Definition 5.2) with the incremental commitment chain (Equation (42)). The trace engine must guarantee completeness: no state change occurs outside the traced pipeline. Replay must be deterministic (Definition 5.4).
Constraint Engine. The constraint derivation function $\mathcal{D}$ (Definition 6.1) must be implemented as a deterministic compiler from SIR to arithmetic constraints. Manual constraint injection is forbidden. The constraint coverage matrix must be maintained as a living artifact, updated whenever constraints, invariants, or transition classes change. The underconstraint analysis (Section 6.3) must be performed as part of the continuous integration pipeline.
Prover and Verifier. The prover generates proofs binding to the full trace (Section 7.1). The verifier implements the strict verification pipeline (Section 8). Both must enforce domain separation (Section 7.2). Version mismatches between constraint system, proof system, and verifier must result in rejection.
Testing Requirements. Mandatory testing includes: differential testing (implementation versus specification), invariant fuzzing (random states and adversarial transitions), constraint

fuzzing (searching for valid constraint satisfaction with invalid semantics), trace mutation testing (modifying traces and verifying detection), witness manipulation (attempting alternate witness assignments), and composition stress testing (simulating multi-system interaction with adversarial patterns).

Performance. Optimizations must not remove constraints, weaken invariants, or introduce nondeterminism. If performance conflicts with correctness, performance loses. This is a non-negotiable design principle.

14 Limitations

VSEL does not claim to solve all problems in verifiable computation. The following limitations are explicitly acknowledged.

Specification completeness. The system's correctness is bounded by the completeness of the formal specification. If the specification fails to capture the intended behavior—through omission, ambiguity, or error—the system can be formally correct and semantically wrong. No formal method eliminates this risk; it can only be bounded through rigorous specification practices and adversarial self-audit. The integration of economic invariants (Section 4.1) reduces but does not eliminate this risk for the economic domain: the economic invariant set is complete only insofar as the domain experts and formal methods engineers have correctly identified all economically relevant properties.

Computational cost. The layered architecture introduces overhead at every level: formal specification maintenance, SIR compilation, constraint derivation, proof generation, and verification. The addition of economic invariants increases constraint system size and proof generation cost. For systems where the cost of formal rigor exceeds the cost of potential failure, VSEL may be disproportionate. The architecture is designed for systems where correctness is non-negotiable.

Refinement proof effort. Establishing the refinement chain (Section 9) requires significant proof engineering effort, estimated at months of work by skilled proof engineers. This is a one-time cost per system version but must be repeated for significant changes.

Hardware and side-channel attacks. VSEL's formal model does not address hardware-level faults or side-channel attacks outside the formal model. If the execution environment is compromised at the hardware level, the formal guarantees do not apply.

Liveness under adversarial conditions. While VSEL ensures safety (no invalid execution is accepted), liveness guarantees (every valid execution can be proven and verified in bounded time) depend on the computational assumptions of the proof system and the availability of the execution environment.

Economic parameter calibration. While VSEL formalizes economic invariants as first-class predicates, the *calibration* of economic parameters (maximum leverage ratios, extraction thresholds, liquidity minimums) requires domain expertise and empirical analysis that formal methods alone cannot provide. The formal system ensures that whatever parameters are chosen are enforced; it does not determine what the correct parameters are. Incorrect calibration can render economic invariants either too permissive (allowing exploitation) or too restrictive (preventing legitimate activity).

15 Related Work

Zero-knowledge virtual machines (zkVMs). Systems such as RISC Zero, SP1, and zkWASM provide general-purpose verifiable computation by compiling programs to arithmetic circuits. These systems verify that a program executed correctly on a virtual machine but do not verify that the program itself correctly implements the intended semantics. The circuit encodes the VM's instruction set, not the application's formal specification. VSEL addresses the layer above: ensuring that the semantics being proven are the intended semantics.

ZK rollups. Rollup systems (zkSync, StarkNet, Polygon zkEVM) use zero-knowledge proofs to verify state transitions of an execution layer. The constraint systems in these architectures are typically hand-written or compiler-generated from the execution engine, without a formally verified derivation from a specification. VSEL's constraint derivation layer (Section 6) and refinement chain (Section 9) address this gap.

Formal verification of circuits. Projects such as Ecne, Picus, and Coda's SNARK verification efforts apply formal methods to verify properties of specific circuits. These efforts are valuable but operate at the circuit level, verifying that a given circuit satisfies certain properties. They do not establish that the circuit is a sound and complete encoding of a formal specification. VSEL's contribution is the end-to-end refinement chain from specification to proof.

Certified compilers. CompCert and similar projects provide formally verified compilation from high-level languages to machine code, establishing simulation relations between source and target semantics. VSEL's refinement strategy (Section 9) is directly inspired by this approach, applying it to the domain of constraint derivation and proof generation rather than traditional compilation.

Refinement-based verification. The refinement calculus, B-method, and Event-B provide frameworks for stepwise refinement of specifications to implementations. VSEL adapts these ideas to the specific challenges of zero-knowledge proof systems, where the "implementation" is not executable code but an arithmetic constraint system, and the correctness criterion is not behavioral equivalence but semantic validity under proof.

Assume-guarantee reasoning. The assume-guarantee paradigm for compositional verification has a long history in concurrent systems verification. VSEL applies this paradigm to the composition of independently proven cryptographic systems (Section 10.1), where the challenge is not concurrency but semantic consistency across system boundaries.

Theorem proving for cryptographic protocols. Tools such as EasyCrypt, CryptoVerif, and the Foundational Cryptography Framework provide machine-checked proofs of cryptographic protocol security. VSEL's cryptographic model (Section 11) relies on standard cryptographic assumptions but does not itself provide machine-checked cryptographic proofs; it defines the interface between the semantic layer and the cryptographic layer.

16 Conclusion

This paper has presented the Verifiable Semantic Execution Layer (VSEL), an architecture that closes the gap between verified execution and correct execution in cryptographic proof systems. The core insight is that correctness must be defined by a formal specification, not by a circuit, and that every component of the system—from execution to constraints to proofs to verification—must be bound to this specification through formally justified refinement relations.

VSEL redefines the correctness criterion from "execution satisfies a circuit" to "execution belongs to the formal language of valid traces induced by the system's specification." This redefinition eliminates an entire class of systemic failures—those in which proofs validate over semantically invalid executions—by ensuring that the constraint system is a sound and complete projection of the formal semantics.

The architecture introduces several mechanisms that are absent from current systems: explicit semantic mappings with commutativity obligations, a systematic underconstraint analysis framework, witness uniqueness requirements, assume-guarantee composition contracts, and a refinement chain connecting five abstraction levels from formal specification to cryptographic proof. The cryptographic model integrates hybrid post-quantum primitives with temporal sensitivity classification and migration protocols for long-term validity.

The primary limitation remains specification completeness: the system is as correct as its formal model. This limitation is inherent to all formal methods and cannot be eliminated, only bounded through rigorous specification practices, adversarial self-audit, and continuous

validation.

Future work includes mechanized proofs of the refinement chain in Lean 4 and Coq, model checking of the abstract specification in TLA+, development of automated underconstraint detection tools, and empirical evaluation of the architecture on concrete protocol implementations.

The system is designed so that incorrect implementations fail early instead of succeeding incorrectly. This is the only standard that matters.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018/046, 2018.

[3] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.

[4] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

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

[6] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[7] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[8] National Institute of Standards and Technology. Post-quantum cryptography standardization. FIPS 203, 204, 205, 2024.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art*. Springer, 2004.

[10] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *CADE*, pages 625–635, 2021.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[12] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptology ePrint Archive*, 2019/953, 2019.

[13] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE S&P*, pages 459–474, 2014.

[14] V. Buterin. An incomplete guide to rollups. https://vitalik.eth.limo/general/2021/01/05/rollup.html, 2021.

[15] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security*, pages 519–535, 2021.