Towards Trustworthy Smart Contract Synthesis: A Multi-Agent Framework with Lean-Based Verification

Bowei Zhang[1]*, **Hanbing Liu**[1]*, **Qixin Tian**[1], **Siyu Chen**[1], **Ziyuan Wang**[1], **Qi Qi**[1,2,3]†
[1]Gaoling School of Artificial Intelligence, Renmin University of China, Beijing, China,
[2]Beijing Key Laboratory of Research on Large Models and Intelligent Governance
[3]Engineering Research Center of Next-Generation Intelligent Search and Recommendation, MOE
{2023201812, liuhanbing, qi.qi}@ruc.edu.cn

Abstract

Smart Contracts are the foundation of Decentralized Finance (DeFi), executing financial logic without trusted intermediaries. Recent advances in large language models (LLMs) have substantially lowered the barrier to smart contract development by enabling code generation from natural language. However, because smart contracts are immutable and directly manage financial assets, this accessibility introduces a critical trust gap: generated contracts are easy to produce but hard to trust. To bridge this gap,

We present **LeVer**, the first trustworthy smart contract synthesis framework that integrates LLM-based generation with **Le**an-based autoformalization and **Ver**ification. LeVer employs a closed-loop multi-agent architecture to iteratively generate, verify, attack, and repair contracts, providing both formal guarantees and empirical robustness. To facilitate the adoption of automated formal verification in smart contract generation and audition, we open-source our framework and datasets at: https://github.com/gl-bowei/LeVer

1 Introduction

Smart Contracts (SC, Khan et al., 2021) serve as the autonomous logic layer of modern blockchains, functioning conceptually as "digital vending machines." Just as a vending machine automatically releases a product when funds are insertedwithout requiring a shopkeepersmart contracts execute complex agreements strictly based on pre-defined code, reducing reliance on trusted intermediaries. Beyond simple transactions, this technology has evolved to underpin the multi-billion dollar Decentralized Finance (DeFi, Jiang et al., 2023) ecosystem, enabling automated lending, trading, and asset management. Furthermore, its application is expanding into real-world sectors, such as automating supply chain logistics and managing digital identities (Casino et al., 2019), fundamentally reshaping how value is exchanged in the digital economy.

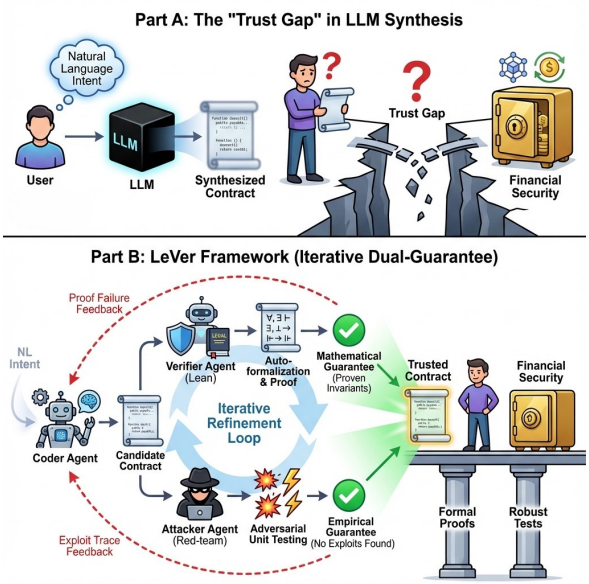


Figure 1: An illustration of the Trust Gap in LLM Smart Contract Synthesis (the upper half) and our Trustworthy LeVer Framework (the lower half).

To democratize access to this transformative technology, Large Language Models (LLMs) have recently emerged as powerful tools for smart contract synthesis (Chatterjee and Ramamurthy, 2024). By automating the translation of natural language intents into executable Solidity code, LLMs significantly lower the development barrier, empowering a broader range of non-experts to participate in the blockchain ecosystem.

However, this automation introduces a critical paradox. Unlike traditional software, smart contracts govern direct financial assets and are immutable once deployed. A single subtle logic flaw or hallucination by the LLM can lead to irreversible financial losses (Nikolic et al., 2018). Consequently, while LLMs solve the accessibility problem, a critical "trust gap" remains. Users are

*Equal contribution.
†Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 39548–39582

left with a dilemma: the code is easy to generate, but dangerously difficult to trust.

Current security measures are ill-equipped to bridge this gap. Traditional rule-based static analysis tools, while widely used, often suffer from high false-positive rates and struggle to capture high-level logic errors (Ghaleb and Pattabiraman, 2020). Manual auditing, though effective, is unscalable and prohibitively expensive for the mass generation of contracts (David et al., 2023). Furthermore, recent attempts to employ LLMs for vulnerability detection typically rely on natural language feedback (Ince et al., 2025). While flexible, these methods lack rigorous standards and deterministic guarantees, often failing to distinguish between a correct implementation and a plausible hallucination. This limitation highlights the urgent need for a verification approach that is not only automated but also mathematically rigorous.

Ideally, the rigorous security standards required for DeFi should be met through Formal Verification (Tolmach et al., 2021). Unlike empirical testing, formal methods (Clarke et al., 1996) treat code as mathematical objects. By rigorously proving that a contract adheres to specific invariants, one can guarantee immunity to entire classes of vulnerabilities (e.g., reentrancy or integer overflows, Zheng et al., 2023b) with mathematical certainty. However, a prohibitive "formalization gap" hinders widespread adoption. Traditional verification relies on specialized languages like Coq or Isabelle and necessitates deep expertise in mathematical logica skill set rare even among experienced developers (Peleska, 2019). Consequently, formal verification has historically remained a manual, labor-intensive process, making it disconnected from the rapid, automated workflow of modern code synthesis.

Recent advancements in neural reasoning offer a turning point, with the Lean theorem prover (Moura and Ullrich, 2021) emerging as a key enabler (Wang et al., 2024a; Jiang et al., 2024). Lean distinguishes itself not only by its ability to verify complex mathematical properties but also as a functional programming language, making it structurally amenable to modern code models. Crucially, recent breakthroughs demonstrate that LLMs exhibit a surprising proficiency in Lean, effectively bridging the chasm between natural language and formal logic. This capability unlocks the potential for "Auto-formalization"a paradigm where LLMs autonomously translate Solidity code into Lean models. By establishing this neuro-symbolic synergy, we can finally harness the mathematical rigor of formal proofs within an automated pipeline, reducing the manual formalization and proof-engineering effort in the synthesis loop.

To instantiate this vision, we introduce a multi-agent framework for trustworthy smart contract synthesis. Our system orchestrates three specialized agents to mimic a human expert team: (1) A CODER AGENT that synthesizes Solidity code and iteratively refines it based on feedback; (2) A VERIFIER AGENT that acts as the "mathematician," extracting essential security invariants (theorems), auto-formalizing Solidity logic into Lean, and constructing machine-checkable proofs; and (3) An ATTACKER AGENT that serves as a "red-teamer" for properties that are formally undecidable or computationally expensive to prove. It generates adversarial transaction sequences to hunt for concrete counterexamples. Crucially, our framework closes the loop: the CODER utilizes the combined feedback logical errors from proof failures and concrete exploits from the ATTACKER to repair the contract. This yields a dual guarantee: mathematical certainty for proven properties and empirical robustness against sophisticated attacks for complex behaviors.

In summary, this paper makes the following contributions:

1. **Auto-Formalization for Smart Contract.** We develop **LeVer**, which to the best of our knowledge, is the first framework to integrate Lean-based auto-formalization into the smart contract generation loop. By lifting the contract safety standard from fuzzy natural language audition feedback to rigorous mathematical proofs, we bridge the *Trust Gap* and enable LLMs to direct reason about high-level economic invariants rather than just syntax.

2. **Dual Guarantee and Iterative Refinement.** The proposed closed-loop architecture where a static Verifier and a dynamic Attacker operate synergistically. Crucially, our Attack-to-Property mechanism converts concrete exploit traces into new formal theorems, enabling the system to iteratively evolve and permanently immunize itself against discovered threats.

3. **Potential Application.** Extensive evaluations on large-scale benchmark demonstrate that LeVer establishes a new state-of-the-art, reducing the adversarial attack success rate to

a negligible 2.4%. Beyond quantitative gains, our work serves as a proof-of-concept for deploying LLMs in zero-tolerance financial domains, beyond just human-assisted copilots.

2 Related Work

SC Vulnerability Detection. Traditional automated analysis primarily relies on static analysis and fuzzing. Symbolic execution tools like Oyente (Luu et al., 2016) and Mythril (Mueller, 2018) explore execution paths via intermediate representations to detect low-level issues (e.g., integer overflows), while successors like Securify (Tsankov et al., 2018) improve coverage using dependency graphs. However, these static methods notoriously suffer from path explosion and high false-positive rates. On the dynamic front, fuzzers like ContractFuzzer (Jiang et al., 2018) and ILF (He et al., 2019) generate random transaction sequences to trigger anomalies. Yet, lacking semantic understanding, they struggle to construct the complex, state-dependent sequences required to uncover deep logic bugs in DeFi protocols. In contrast, our framework employs a strategic Attacker Agent that utilizes memory and planning to actively hunt for deep exploits, moving beyond blind fuzzing.

Formal Methods for SC. Formal verification aims to mathematically guarantee correctness through model checking or theorem proving. Model checkers (e.g., VeriSol, Bhargavan et al., 2016; NuSMV, Cimatti et al., 2000) verify state properties but face state-space explosion in complex contracts. Theorem proving offers the highest security standard; projects like KEVM (Hildenbrandt et al., 2018) and Coq-based frameworks (Annenkov et al., 2020) define rigorous semantics for the EVM. Despite their completeness, these methods face a prohibitive "formalization gap": translating Solidity to formal specifications requires niche expertise and manual effort, preventing widespread adoption. We address this by leveraging LLMs to automate the translation from Solidity to Lean 4, enabling rigorous verification within a fully automated synthesis pipeline.

LLM applications in SC. Recent works leverage LLMs for both synthesis and auditing. In synthesis, while general LLMs handle standard templates well, they falter on complex logic. Approaches like FSM-SCG (Luo et al., 2025) and SmartCoder (Yu et al., 2025) introduce structural constraints (e.g., finite state machines) or security-aware reinforcement learning to improve reliability. In auditing, multi-agent frameworks (e.g., LLM-SmartAudit, Wei et al., 2024) and neuro-symbolic tools (e.g., PropertyGPT, Liu et al., 2024; SmartInv, Wang et al., 2024b) integrate semantic understanding to reduce false positives or infer invariants. However, these solutions remain largely probabilistic or require human-in-the-loop validation. Our work distinguishes itself by establishing a closed feedback loop where code is not merely checked by another hallucination-prone LLM, but rigorously proved by a theorem prover and stress-tested by adversarial agents.

3 Preliminaries

The core challenge in trusted code generation lies in bridging the semantic gap between the imperative execution of the Ethereum Virtual Machine (EVM) and the declarative rigor of formal logic. To enable neuro-symbolic reasoning, we must first lift smart contracts from code to mathematical objects. In this section, we formalize the execution model of smart contracts (Section 3.1), introduce the proof-synthesis paradigm in Lean 4 (Section 3.2), and finally reformulate the synthesis task as a joint property-inference and contract-synthesis objective (Section 3.3).

3.1 Smart Contract

A smart contract is an autonomous program deployed on a blockchain that functions as a replicated state machine. Here we abstract its execution model into three core components that align with the functional definitions used in theorem proving:

Persistent State (Σ). Contracts maintain permanent on-chain storage (e.g., a ledger mapping users to balances). We denote the set of all possible storage configurations as the State Space $\Sigma_{\mathcal{C}}$, where a specific snapshot at time t is an instance $\sigma \in \Sigma_{\mathcal{C}}$.

Execution Context (Γ). Unlike standalone programs, contract execution depends heavily on the external execution context (e.g., `msg.sender`, `block.timestamp`). We encapsulate these environmental variables into a Context Space Γ, where a specific environment is denoted as $\gamma \in \Gamma$.

Logic as Transition Mapping (Υ). The business logic consists of executable functions (e.g., `pay`, `withdraw`) triggered by user transaction inputs $\iota \in I_{\mathcal{C}}$, where $I_{\mathcal{C}}$ encodes both the invoked method and its arguments. We abstract this logic

as a deterministic State Transition Function $\Upsilon_{\mathcal{C}}$. It takes the current snapshot $\sigma \in \Sigma_{\mathcal{C}}$, the environmental context $\gamma \in \Gamma$, and the user input $\iota \in I_{\mathcal{C}}$ to compute an execution result:

$$\Upsilon_{\mathcal{C}} : \Sigma_{\mathcal{C}} \times \Gamma \times I_{\mathcal{C}} \to \mathcal{R}_{\mathcal{C}}.$$

A successful execution may update the contract state, emit events, and specify asset transfers. We therefore represent a successful outcome by an abstract output type $O_{\mathcal{C}}$, whose state component is written as $\text{out.state} \in \Sigma_{\mathcal{C}}$, and define the result type as:

$$\mathcal{R}_{\mathcal{C}} ::= \text{success}(\text{out} : O_{\mathcal{C}}) \mid \text{failure}(\text{err} : \text{Error}).$$

This modeling captures the crucial property of *Atomicity*: if a security check fails (e.g., insufficient balance), the transaction reverts immediately, and no partial or invalid state is committed to the ledger.

3.2 Formal Verification in Lean

Lean is an interactive theorem prover where mathematical statements are represented as types, and proofs are executable programs. We briefly explain how it enables LLMs to perform verification.

Proof as Program Synthesis. Analogous to standard programming where a function body must match its signature, Lean treats a mathematical theorem as a Type Signature Θ, and its proof as a Program Term t implementing that signature ($t : \Theta$). Once a candidate term t is produced, proof checking is reduced to compilation/type checking: if the term t type-checks against Θ, the theorem is proven. This paradigm effectively transforms formal proof checking into a standard code-validation task, enabling LLMs to synthesize mathematical proofs by generating valid proof terms and tactics.

Semantic Translation. To apply this to smart contracts, under the deterministic, closed-system abstraction adopted in this paper, we define a Semantic Translation Operator $\mathcal{T}$. This operator maps the imperative Solidity code $\mathcal{C}$ into the functional definitions required by Lean:

$$\mathcal{T}(\mathcal{C}) \mapsto (\Sigma_{\mathcal{C}}, \Gamma, I_{\mathcal{C}}, \Upsilon_{\mathcal{C}}),$$

where $\Upsilon_{\mathcal{C}}$ is the aggregate state transition function defined in Section 3.1. It encapsulates the logic of all individual contract methods (e.g., `deposit`, `withdraw`), dispatching execution to the specific logical branch that matches the user input $\iota \in I_{\mathcal{C}}$. The output structure $O_{\mathcal{C}}$ reifies side effects such as emitted events and transfer intents into explicit data so that they can be reasoned about inside Lean.

By reifying the code into this single functional entry point, we transform the verification of behavioral requirements into a consistent proof-checking problem. The concrete translation rules are detailed in Appendix A.2.

3.3 Problem Formulation

Departing from the traditional paradigm of retrospective vulnerability detection, we formulate *Trustworthy Smart Contract Synthesis* as a proactive property satisfaction problem over state-based safety properties.

Logical Exclusion of Vulnerabilities. Instead of enumerating known vulnerability patterns V (e.g., `reentrancy`, `overflow`), we approach from the sub-universe of safety invariants $\Phi \subseteq \mathbb{P}$ that the ideal contract can satisfy. For the state-based vulnerabilities considered in this section, we assume a covering relation between properties and vulnerabilities: for every vulnerability $v \in V$, there exists a subset of invariants $\Phi_v \subseteq \Phi$ such that satisfying Φ_v serves as a sufficient condition for excluding v, i.e.,

$$\forall v \in V,\ \exists \Phi_v \subseteq \Phi,\ \left(\bigwedge_{\phi \in \Phi_v} \phi \right) \Rightarrow \neg v.$$

For example, a solvency invariant ϕ may mandate that the contract's vault balance must equal the sum of user deposits. If ϕ is preserved across all reachable states, then attacks that rely on inconsistent accounting states are ruled out by construction. Thus, we can focus on establishing ϕ rather than enumerating individual exploit patterns.

Optimization Objective. In practice, deriving the complete property set $\Phi(R)$ from users' informal functional requirements R is non-trivial. Therefore, a trustworthy synthesis framework must not only generate code but also infer an approximate covering set $\hat{\Phi}(R)$ that bridges the semantic gap between vague intent and rigorous logic.

We then define the *Trustworthy Smart Contract Synthesis Task* as follows. Given requirement R, the goal is to infer a property set $\hat{\Phi}(R) \subseteq \mathbb{P}$, and to generate a valid contract $\mathcal{C}^* \in \mathbb{C}$ that maximizes the number of verified properties:

$$\mathcal{C}^*(R) = \arg\max_{\mathcal{C} \in \mathbb{C}} \sum_{\phi \in \hat{\Phi}(R)} \mathbb{I}(\text{Verify}(\mathcal{C}, \phi)).$$

To define $\text{Verify}(\mathcal{C}, \phi)$, let $\text{Reach}_{\mathcal{C}}(\sigma)$ denote that σ is reachable from the initial state of contract $\mathcal{C}$ under $\Upsilon_{\mathcal{C}}$. We define

$$\begin{aligned}\text{Verify}(\mathcal{C}, \phi) := \forall \sigma, \gamma, \iota, o, \\ \big(\text{Reach}_{\mathcal{C}}(\sigma) \wedge \\ \Upsilon_{\mathcal{C}}(\sigma, \gamma, \iota) = \text{success}(o)\big) \\ \Rightarrow \phi(o.\text{state}).\end{aligned}$$

This condition asserts that for any reachable state σ, external environment γ, and user input ι, if the transaction executes successfully (does not revert), then the resulting post-state must rigorously satisfy the invariant ϕ. When ϕ also holds in the initial state, this establishes ϕ as an inductive invariant over all reachable states.

This formulation acknowledges that trustworthiness is a spectrum. We aim to automatically construct and expand the covering property set $\hat{\Phi}(R)$, then iteratively refine $\mathcal{C}$ to asymptotically approach the regime where $\forall \phi \in \Phi(R), \text{Verify}(\mathcal{C}, \phi)$ holds.

4 The LeVer Framework

We now present **LeVer**, a multi-agent framework that bridges the gap between automated synthesis and rigorous security assurance by orchestrating a closed-loop refinement workflow. As illustrated in Figure 2, LeVer integrates generative models with neuro-symbolic formal verification and adversarial agent simulation to iteratively evolve smart contracts. This section provides a high-level overview of the pipeline, followed by a detailed breakdown of the core assurance mechanisms. Design details are provided in Appendix A.

4.1 Overview

The workflow initiates with a user requirement R, which the **Coder** translates into an initial Solidity contract $\mathcal{C}_0$. Subsequently, the system enters an iterative refinement loop indexed by k. In each iteration, the current candidate contract $\mathcal{C}_k$ undergoes a dual-assurance process.

The **Verifier** (see Section 4.2) lifts the code into the Lean 4 domain to mathematically certify safety properties, guaranteeing that specific classes of vulnerabilities are theoretically impossible.

Complementarily, the **Attacker** (see Section 4.3) instantiates a sandbox environment to hunt for counter-examples against properties that are undecidable or computationally expensive to prove.

The outputs from both engines, i.e. failed proofs and successful attack traces, are synthesized into structured feedback $\mathcal{F}_k$, which guides the Coder to generate an improved version $\mathcal{C}_{k+1}$. This process terminates when all properties are verified or the contract survives a predefined threshold of adversarial episodes, or maximum round is reached.

4.2 The Neuro-Symbolic Verifier

The Verifier certifies the trustworthiness of the candidate contract $\mathcal{C}_k$ by constructing mathematical proofs in Lean 4. It operates through three pipelined sub-modules:

PropertySelector. This module bridges the semantic gap between informal requirements and formal specifications. To construct the approximate covering set $\hat{\Phi}(R)$, the Selector utilizes a *Hierarchical Assurance Spectrum* (detailed in Appendix A.1). Instead of a flat list, this spectrum organizes properties along a progressive gradient of abstraction spanning from foundational execution integrity (e.g., overflow prevention, memory safety) up to high-level economic behavior insurance (e.g., game-theoretic robustness, incentive compatibility). Initially, it retrieves a relevant subset of invariants based on the semantics of the initial contract $\mathcal{C}_0$. Furthermore, the module supports dynamic evolution via an *Attack-to-Property* mechanism: novel properties ϕ', distilled from adversarial attack traces by the Analyst (see Section 4.3), are injected into the spectrum. This ensures that the formal specification $\hat{\Phi}(R)$ evolves adaptively alongside the code refinement.

LeanFormalizer. Following selection, this module executes the auto-formalization task by applying the Semantic Translation Operator $\mathcal{T}$. Under the deterministic, closed-system abstraction adopted in this paper, it maps the imperative Solidity code $\mathcal{C}_k$ into the functional Lean model $(\Sigma_{\mathcal{C}_k}, \Gamma, I_{\mathcal{C}_k}, \Upsilon_{\mathcal{C}_k})$, transforming mutable storage into immutable structures, function logic into state transitions, and side effects into explicit output data. To prepare for verification, it instantiates a theorem template for each property $\phi \in \hat{\Phi}(R)$ using the `sorry` tactic as a temporary placeholder:

$$\begin{aligned}\forall \sigma, \gamma, \iota, o, \\ \big(\text{Reach}_{\mathcal{C}_k}(\sigma) \wedge \\ \Upsilon_{\mathcal{C}_k}(\sigma, \gamma, \iota) = \text{success}(o)\big) \\ \Rightarrow \phi(o.\text{state}) \ :=\ \texttt{sorry}.\end{aligned}$$

This step ensures that the formal structure is syntactically valid before the proof generation phase

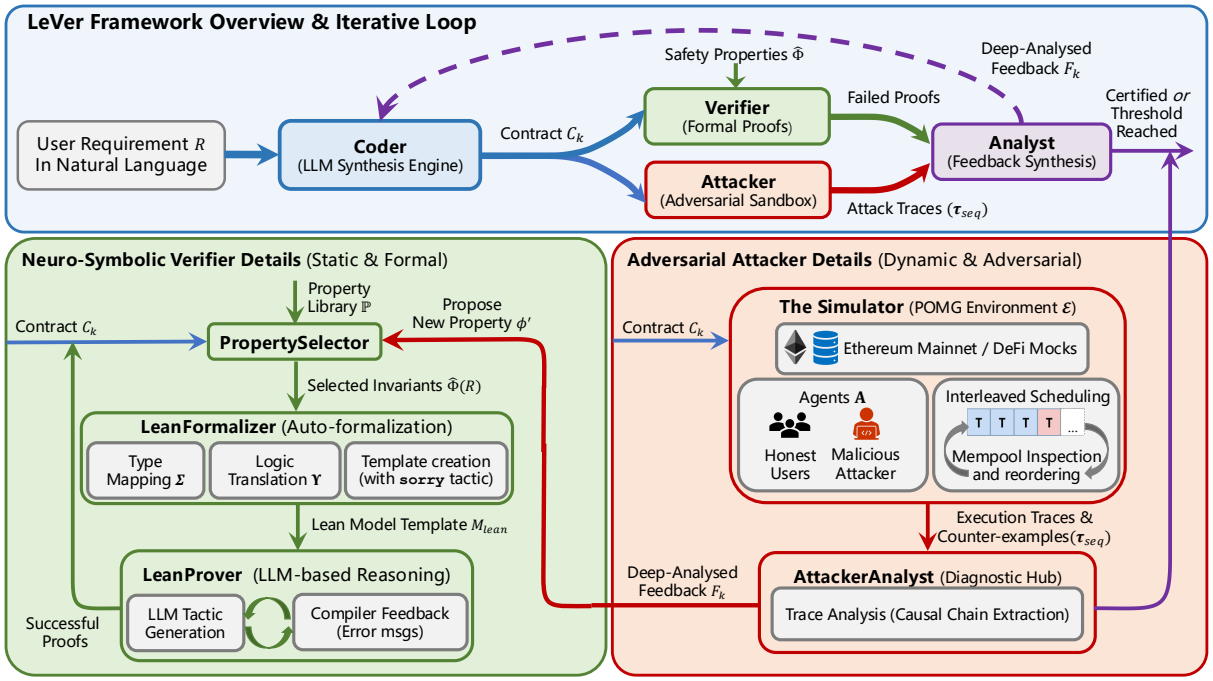


Figure 2: Overview of the **LeVer** framework. The top panel shows the **iterative synthesisanalysis loop (top)**: a Coder generates a contract candidate, which is jointly examined by a neuro-symbolic Verifier and an adversarial Attacker, and their signals are consolidated to guide subsequent refinement. The bottom panels detail the two engines: **The Verifier (bottom left)** automatically formalizes contracts in Lean and searches for formal proofs of safety properties, while **The Attacker (bottom right)** simulates adversarial executions to uncover concrete counterexamples. Crucially, attack-induced traces are fed back to expand verification properties, enabling mutual reinforcement between dynamic attacks and static formal reasoning.

begins. It is noteworthy that this translation operator $\mathcal{T}$ is non-trivial to implement; detailed rule design and a concrete example can be found in Appendix A.2 and Appendix A.3, respectively.

LeanProver. Finally, this module attempts to replace the `sorry` placeholders in each theorem with valid logical arguments, completing the proofs. Operating within Lean's verification environment, it utilizes reasoning capabilities to synthesize the necessary proof tactics. The process is guided by the compilers feedback, which serves as critical signals for the agent to debug and refine its strategy. A completed proof provides a mathematical guarantee that C_k preserves the property ϕ under all possible inputs, effectively precluding the corresponding class of vulnerabilities.

4.3 The Adversarial Attacker

While formal verification offers theoretical guarantees, computational constraints often leave complex properties indeterminate. The Attacker addresses this by subjecting the candidate contract C_k to dynamic, worst-case execution scenarios, attempting to find a counterexample that proves the property is violated and providing feedback to help the coder refine the contract.

The Simulator. We model the execution environment $\mathcal{E}$ as a Partially Observable Markov Game (POMG). The Simulator deploys C_k alongside heterogeneous agents $\mathbf{A}$: Honest Agents A_{honest} simulate organic user behaviors to drive standard state transitions, while Adversarial Agents $A_{attacker}$ aim to exploit the system. Based on their partial observation of the public chain state and private history, these adversaries strategically select the next malicious transaction τ_{attack} to execute. To detect concurrency bugs, the simulator employs an *Adversarial Scheduling* mechanism, granting attackers the privilege to inspect the memory pool and manipulate transaction ordering, sandwiching honest transactions to efficiently explore the combinatorial space of state inconsistencies. See Appendix A.5 for design details and the algorithm pseudo-code.

AttackAnalyst. This module acts as the diagnostic hub. Instead of heuristic monitoring, it employs *Runtime Oracles* $\mathcal{O}$ embedded in the sandbox to rigorously enforce safety invariants (See Appendix A.5 for the formal definition). These

oracles check the contract state after every atomic transition; if a violation is triggered (e.g., a balance inconsistency), the Analyst deterministically captures the exploit trace τ_{seq}. This trace is then converted into feedback $\mathcal{F}_k$ to guide the Coder in patching the vulnerability, while the semantic pattern of the attack is distilled into new formal properties ϕ' for the Verifier, ensuring the specific threat is permanently ruled out in future iterations.

With the architectural framework established, we next outline the experimental setup to evaluate LeVer's empirical performance.

5 Experiments

5.1 Setup

Dataset. We utilize the Smart Contract Generation Benchmark (Luo et al., 2025), comprising over 30,000 reverse-engineered requirement-contract pairs collected from Etherscan.

Baselines. We compare LeVer against two paradigms: (1) Direct (Chatterjee and Ramamurthy, 2024), the unguided generation; and (2) FSM (Luo et al., 2025), the state-of-the-art (SOTA) smart contract generation method using Finite State Machines as intermediate representations.

Backbones. Evaluations are conducted on four frontier foundation models: Gemini 3 Pro (Google DeepMind, 2025), GPT-5 (OpenAI, 2025), Qwen3-Max (Qwen Team, 2025), and DeepSeek-V3.2 (DeepSeek-AI, 2025).

Metrics. We employ a comprehensive suite of metrics covering both standard correctness and trustworthiness. (1) **Standard Correctness.** We report the Pass Rate for static analysis (Slither, no high-severity flags, Feist et al., 2019) and dynamic execution (Foundry, Paradigm, 2021), representing standard auditing strategies. (2) **Trustworthiness Assurance.** We introduce three metrics to assess formal and adversarial robustness:

a) **Formal Property Verification Rate (VR ↑):** The ratio of formally proven properties to the total generated specifications: VR $= |\hat{\Phi}_{proven}|/|\hat{\Phi}_{total}|$.

b) **Average Verified Properties (Avg. ↑):** The average number of verified properties that synthesized contracts have: VR $= |\hat{\Phi}_{proven}|/N$.

c) **Dynamic Attack Success Rate (AR ↓):** The ratio of adversarial transaction tests (N_{test}) that successfully breach safety rules (N_{breach}) in the sandbox: AR $= N_{breach}/N_{test}$.

d) **LLM Audit Pass Rate (PR ↑):** Adopting the LLM-as-a-Judge paradigm (Zheng et al., 2023a), we employ an external LLM to audit generated contracts. Specifically, we adapt the auditing prompt template from iAudit (Ma et al., 2025) for Claude 4.5 Opus and employ a majority voting@5 strategy to determine the final audit result. PR denotes the percentage of contracts where the auditor reports zero high-severity vulnerabilities.

5.2 Main Results

Overall Superiority. Table 1 presents the comprehensive evaluation across four foundation models. LeVer consistently delivers superior performance, establishing a new state-of-the-art benchmark for trustworthy generation.

- **Trustworthy Assurance:** LeVer demonstrates a dominant advantage in reducing vulnerability risks. On Gemini-3-Pro, LeVer suppresses the Attack Success Rate (AR) from 29.0% (Direct) and 34.2% (FSM) to a negligible **2.4%**. This indicates that nearly all economic exploits found in baselines are effectively blocked. Simultaneously, the Verification Rate (VR) surges to **74.1%**, with the average number of verified properties (**Avg.**) increasing from 5.21 to **6.93**. This confirms that LeVer not only proves *more* properties (quantity) but also secures the code against *harder* attacks (quality).

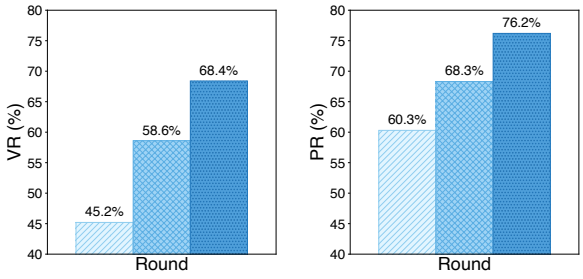
- **Standard Correctness:** Beyond formal guarantees, LeVer also excels in standard metrics. It achieves the highest execution pass rates (Foundry) across all backbones (e.g., **96.9%** on GPT-5), ensuring that the rigorous security constraints do not compromise the functional utility of the contracts.

- **Closing the Source Gap:** Notably, open-source models augmented with LeVer significantly outperform unassisted proprietary models. For instance, DeepSeek V3.2 + LeVer achieves an AR of **9.3%**, which is far superior to GPT-5 Direct (34.3%) and Gemini Direct (29.0%). This suggests that our neuro-symbolic framework effectively bridges the capability gap between model sizes.

Iterative Refinement Capabilities. To analyze the dynamics of the refinement loop, we focus on a subset of "hard instances", defined as tasks necessitating more than three rounds to converge. Figure 3 illustrates the round-wise evolution of as-

Method	Trustworthy Assurance				Std. Correctness	
	VR ↑	Avg.↑	AR ↓	PR ↑	Slither ↑	Foundry ↑
Open-Source Models						
DeepseekV3.2	60.3	4.13	47.3	54.7	61.3	82.7
+ FSM	49.6	4.93	42.3	56.7	56.3	85.3
+ LeVer (Ours)	**77.2**	**6.87**	**9.3**	**84.2**	**76.3**	**89.3**
Qwen3-Max	60.8	2.07	63.7	41.3	61.3	83.8
+ FSM	48.1	2.47	60.3	46.7	53.3	84.0
+ LeVer (Ours)	**78.0**	**6.60**	**13.2**	**82.7**	**73.6**	**90.7**
Proprietary Models						
GPT-5	65.9	5.48	34.3	64.3	67.2	84.4
+ FSM	59.6	5.32	30.7	68.7	72.3	86.2
+ LeVer (Ours)	**73.5**	**6.49**	**5.3**	**91.0**	**80.5**	**96.9**
Gemini-3-Pro	51.4	5.21	29.0	65.9	67.7	83.6
+ FSM	61.1	5.23	34.2	71.3	77.2	88.2
+ LeVer (Ours)	**74.1**	**6.93**	**2.4**	**88.7**	**83.0**	**94.7**
+ LeVer w/o Verifier	68.4	4.77	10.0	76.6	77.3	86.2
+ LeVer w/o Attacker	81.5	6.67	23.3	83.3	80.2	89.2

Table 1: Main results on the Smart Contract Generation Benchmark (Luo et al., 2025). We compare **LeVer** (highlighted in light green) against Direct and FSM baselines across four frontier foundation models. The evaluation covers **Trustworthy Assurance**: VR (Verification Rate, %), Avg. (Average Verified Properties), AR (Attack Success Rate, %), PR (Audit Pass Rate, %) and **Standard Correctness** (Slither, Foundry; %). The bottom rows (highlighted in gray) present ablation studies showing the synergies of the Verifier and Attacker components.



(a) Verification Rate (↑) (b) Pass Rate (↑)

Figure 3: Iterative performance improvement of the **LeVer** framework on challenging smart contract synthesis tasks that require more than three rounds of refinement (Gemini-3 Pro). The figure reports the round-wise evolution of VR and PR, demonstrating consistent gains across successive iterations.

surance metrics for these challenging cases. We observe a strictly monotonic improvement across iterations. Specifically, the Verification Rate (VR) experiences a dramatic surge, climbing from 45.2% in the first round to 68.4% by the third rounda relative improvement of over 51%. Simultaneously, the Pass Rate (PR) steadily ascends from 60.32% to 76.19%. This trajectory indicates a clear two-phase optimization: early iterations effectively resolve syntax errors and standard vulnerabilities (filtering out "low-hanging fruit"), while subsequent rounds allow the neuro-symbolic feedback loop to tackle complex logic bugs and indeterminate properties. The consistent upward trend confirms that LeVer does not merely perturb the code stochastically but actively drives it towards a verifiable fixed point with higher quality.

Case Study. To provide a intuitive demonstration of the iterative refinement process in LeVer, we refer the reader to Appendix B and Appendix C, which present case studies on an Curve-Style Zap-In and an auction contract, respectively.

5.3 Ablation Study

To disentangle the contributions of the Neuro-Symbolic Verifier and the Adversarial Attacker, we conduct ablation studies using the Gemini-3-Pro backbone (bottom rows of Table 1).

Impact of the Verifier (The Safety Floor). Removing the formal verification module (*w/o Verifier*) leads to a degradation across all trustworthiness metrics. The AR rises from 2.4% to 10.0%,

and the VR drops to 68.4%. This highlights that while LLMs can write code that looks correct, without mathematical proofs, they struggle to guarantee strict adherence to invariants (e.g., arithmetic boundaries), leaving "blind spots" that static analysis misses but formal proofs would have caught.

Impact of the Attacker (The Security Ceiling). The exclusion of the adversarial simulation (*w/o Attacker*) reveals a critical insight. Counterintuitively, the VR actually increases to 81.5% (higher than the full model's 74.1%). However, this is a "false sense of security": the AR explodes to 23.3%, nearly reverting to the baseline level.

Without the Attacker to discover complex, dynamic exploits (e.g., sandwich attacks) and translate them into new, challenging properties, the Verifier operates only on the initial, simpler property set. The system easily proves these "easy" theorems (hence the high VR), but fails to defend against the sophisticated threats that matter most.

This result strongly validates the synergy: The Attacker raises the ceiling by injecting necessary difficulty (hard properties), while the Verifier ensures the floor by rigorously proving them.

6 Conclusion

In this work, we address the trust gap in LLM-based smart contract synthesis by proposing LeVer, a closed-loop multi-agent framework that integrates LLM generation with Lean-based autoformalization and formal verification. By coordinating a Coder, a Neuro-Symbolic Verifier, and an Adversarial Attacker, LeVer enables contracts to be iteratively generated, verified, attacked, and repaired within a unified pipeline.

Experiments across multiple backbone models and large-scale benchmarks show that LeVer significantly improves verification rates, reduces attack success, and outperforms strong baselines in trustworthy contract generation. Our results demonstrate that neuro-symbolic methods are not only conceptually appealing but also practically effective for securing smart contracts and DeFi applications. We hope this work can help promote the adoption of neuro-symbolic frameworks as usable and reliable components in automated smart contract development.

7 Limitations

While LeVer demonstrates a significant leap in trustworthiness of smart contract synthesis, we acknowledge three primary limitations that define the boundary of our current capabilities and point towards future research directions.

Semantic Faithfulness of Translation. A prominent limitation lies in the semantic gap between the original Solidity program and its translated Lean code. Our formal guarantees are established over the Lean representation produced by the auto-formalization pipeline, and therefore critically depend on the semantic faithfulness of this translation. While our supplementary evaluations (96.7% pass rate under expert evaluation and 97.4% pass rate under execution cross-check on 300 randomly sampled contract generation tasks) provide strong empirical evidence that the translation is usually well aligned with the original Solidity behavior, *this should not be interpreted as a formal proof of semantic preservation*. In particular, subtle mismatches may still arise in edge cases involving low-level EVM behaviors, external call conventions, event/logging semantics, or rollback-related effects. Importantly, our current evidence suggests that these discrepancies do not affect the core state transitions and safety invariants targeted in this paper, but we fully acknowledge that end-to-end equivalence between Solidity and Lean is not yet guaranteed. Developing a semantics-preserving translation framework, or formally certifying the translation process itself, remains an important direction for future research.

The Gap between Validity and Provability. Our evaluation reveals a residual gap between properties that are intuitively true and those the automated prover can formally certify. Currently, LeVer adopts a full-generation with feedback strategy. While efficient, this approach occasionally leads to false negatives, i.e., the agent fails to find the proof path for a valid theorem. We hypothesize that a fine-grained, interactive proof synthesis approach could bridge this gap. By allowing the agent to observe the granular *Tactic State* (goals and hypotheses) after every single tactic step, the system could achieve higher success rates in proving complex arithmetic and logical invariants. We leave the integration of such step-wise interaction as immediate future work.

Scope of Formalization. Our current semantic translation operator $\mathcal{T}$ models the contract as a deterministic, closed system. Although this aligns with mainstream datasets (e.g., SmartInv, VeriSmart) which treat contracts as isolated units with mocked external contexts, it potentially limits

LeVer's applicability in two real-world scenarios: *Cross-Contract Composability.* DeFi exploits often involve complex interactions between mutable external contracts (e.g., Flash Loans). Our current model does not formalize the recursive state changes of the global call graph. *Stochastic Logic.* Contracts relying on randomness (e.g., lotteries using `block.prevrandao` or Chainlink VRF) introduce non-deterministic state transitions. Since our transition function Υ is strictly deterministic ($\Sigma \times \Gamma \times I \to \mathcal{R}$), it cannot natively reason about probabilistic properties (e.g., "fairness of a draw").

Extending our framework to support probabilistic model checking and global state modeling represents a grand challenge for the next stage of trustworthy synthesis.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No.62472428), Public Computing Cloud, Renmin University of China, the fund for building world-class universities (disciplines) of Renmin University of China, the Research Funds of Renmin University of China (RUC25QSDL123).

References

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. Concert: a smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 215–228.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96.

Fran Casino, Thomas K. Dasaklis, and Constantinos Patsakis. 2019. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics*, 36:55–81.

Siddhartha Chatterjee and Bina Ramamurthy. 2024. Efficacy of various large language models in generating smart contracts. *arXiv preprint arXiv:2407.11019*.

Alessandro Cimatti, Edmund Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2000. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425.

Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, E. Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas A. Henzinger, Gerard J. Holzmann, Cliff B. Jones, Robert P. Kurshan, Nancy G. Leveson, Kenneth L. McMillan, J. Strother Moore, Doron A. Peled, Amir Pnueli, and 8 others. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643.

Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*. Discusses the high time and cost demands of traditional manual smart contract auditing and motivates automated approaches.

DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

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

Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–13. Empirical evaluation showing that popular static analysis tools report many false positives and miss complex bugs.

Google DeepMind. 2025. Gemini 3 Pro Model Card. Model card.

Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548.

Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217.

Peter Ince, Jiangshan Yu, Joseph K. Liu, and Xiaoning Du. 2025. Generative large language model usage in smart contract vulnerability detection. *arXiv preprint arXiv:2504.04685*.

Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, pages 259–269.

Dongwei Jiang, Marcio Fonseca, and Shay B. Cohen. 2024. Leanreasoner: Boosting complex logical reasoning with lean. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Erya Jiang, Bo Qin, Qin Wang, Zhipeng Wang, Qianhong Wu, Jian Weng, Xinyu Li, Chenyang Wang, Yuhang Ding, and Yanran Zhang. 2023. Decentralized finance (defi): A survey. *arXiv preprint arXiv:2308.05282*. Comprehensive survey of DeFi applications, risks, and future research directions.

Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. 2021. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Networking and Applications*, 14:2901–2925.

Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*.

Hao Luo, Yuhao Lin, Xiao Yan, Xintong Hu, Yuxiang Wang, Qiming Zeng, Hao Wang, and Jiawei Jiang. 2025. Guiding llm-based smart contract generation with finite state machine. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 5869–5877.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269.

Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2025. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)*, pages 1742–1754. IEEE Computer Society.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer.

Bernhard Mueller. 2018. Mythril: Security analysis tool for Ethereum smart contracts. In *Black Hat USA*. GitHub repository.

Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 653–667. Demonstrates real-world smart contract vulnerabilities leading to irreversible loss, highlighting immutability risks.

OpenAI. 2025. GPT-5 System Card. System card.

Paradigm. 2021. Foundry: A blazing fast, portable and modular toolkit for ethereum application development. https://github.com/foundry-rs/foundry.

Jan Peleska. 2019. Formal methods: A roadmap for industrial adoption. *ACM Computing Surveys*, 52(4):1–30. Discusses the challenges of traditional formal methods, including reliance on proof assistants like Coq/Isabelle and the scarcity of expertise among practitioners.

Qwen Team. 2025. Qwen3-Max-Instruct. Official blog post.

Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. 2021. Formal analysis of composable defi protocols. In *International Conference on Financial Cryptography and Data Security*, pages 149–161. Springer.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82.

Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. 2024a. Theoreml-lama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*.

Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024b. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2217–2235. IEEE.

Zhiyuan Wei, Jing Sun, Zijiang Zhang, Xianhao Zhang, Meng Li, and Zhe Hou. 2024. Llm-smartaudit: Advanced smart contract vulnerability detection. *arXiv preprint arXiv:2410.09381*.

Lei Yu, Jingyuan Zhang, Xin Wang, Jiajia Ma, Li Yang, and Fengjun Zhang. 2025. SmartCoder-R1: Towards secure and explainable smart contract generation with security-aware group relative policy optimization. *arXiv preprint arXiv:2509.09942*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*.

Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023b. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. *arXiv preprint arXiv:2303.13770*. Empirical study showing reentrancy as a key smart contract vulnerability causing significant financial loss and the limitations of existing detection tools.

A LeVer Design Details

A.1 The Hierarchical Assurance Spectrum

As categorized by Wang et al. (2024b) in *SmartInv*, smart contract vulnerabilities fall into two distinct classes: Generic Vulnerabilities (e.g., reentrancy, integer overflows) stemming from EVM mechanics, and Business Logic Vulnerabilities (e.g., price manipulation, privilege escalation) arising from specific application contexts.

This inspired us to organize the property space Φ into a Hierarchical Assurance Spectrum. This spectrum spans four abstraction levels, progressively bridging the gap from low-level code safety to high-level mechanism design. Importantly, this hierarchy classifies assurance targets rather than the expressive power of Lean itself.

Level 1: Execution and Language Safety.

This level defines the "physical laws" of the EVM execution environment. These properties are typically context-light and universally applicable.

- *Arithmetic Integrity:* Beyond standard overflow checks, this includes verifying precision safety (e.g., enforcing "multiply before divide"), preventing silent truncation during type casting, and handling signed integer boundaries.
- *Resource & Memory Safety:* Constraints to prevent Gas Exhaustion (DoS) due to unbounded loops, ensure correct storage pointer initialization to avoid "ghost variable" collisions, and validate stack depth.
- *Control Flow Integrity:* Enforcing the Checks-Effects-Interactions (CEI) pattern to structurally block reentrancy, and ensuring strict adherence to standard interfaces (e.g., ERC-20/721) to prevent unexpected reverts due to selector clashes.

Level 2: Functional Correctness.

Viewing the contract as a Finite State Machine (FSM), this level ensures the code faithfully implements the developer's design intent.

- *Transition Validity:* The state transition graph must be closed and legal (e.g., a "Refunded" order cannot revert to "Active"). Mutual exclusion locks (Mutex) must ensure assets cannot exist in dual states.
- *Access Control:* Privileges must be validated against correct historical snapshots (defending against Flashloan Governance Attacks) and adhere to the principle of least privilege.
- *Hoare Logic Specifications:* For complex business logic (e.g., a Dutch Auction price decay formula), we verify that specific preconditions imply the intended post-conditions.

Level 3: Temporal, Liveness and Interaction Safety.

Moving beyond static correctness, this level focuses on system evolution over time in an adversarial environment.

- *Liveness & Deadlock Freedom:* The system must guarantee progress, such as ensuring that users can eventually withdraw funds (Guaranteed Withdrawal) regardless of administrator actions or oracle failures.
- *Interaction Safety:* Enforcing "Pull-over-Push" payment patterns to prevent DoS attacks where malicious receivers block transfers.
- *Transaction Ordering Independence (TOI):* Critical operations must be resilient to sequence manipulation, ensuring that Front-running by miners or bots cannot violate the fairness principle (e.g., "First-Come-First-Served").

Level 4: Economic and Mechanism Design.

The highest level treats the contract as a game-theoretic system involving rational agents maximizing their utility.

- *Incentive Compatibility:* Proving that honest participation is a Dominant Strategy (or another desired equilibrium notion), ensuring mechanisms have sufficient incentives (covering Gas/opportunity costs) and credible slashing penalties.
- *Strategy Resistance:* Systems must resist Sybil attacks and Commit-Reveal schemes must prevent value manipulation during the reveal phase.
- *Market Efficiency (DeFi Specific):* Verifying properties like the monotonicity of pricing curves (AMMs) to eliminate risk-free arbitrage loops and ensuring protection against "Tyranny of the Majority" in governance (e.g., Ragequit mechanisms).

A.2 Semantic Translation Rules

To reason about smart contracts within a rigorous mathematical framework, we design and implement a direct semantic mapping mechanism. Smart contracts are inherently imperative, state-mutating programs running on the EVM. Verifying such code is challenging due to implicit global states and side effects. Therefore, under the deterministic, closed-system abstraction adopted in this paper, our modeling objective is to reconstruct contract semantics at the level of Type Theory as purely functional Lean models suitable for proof search and type checking. For a fixed contract $\mathcal{C}$, this translation yields the functional interface $(\Sigma_{\mathcal{C}}, \Gamma, I_{\mathcal{C}}, \Upsilon_{\mathcal{C}})$. In this subsection, we elaborate the execution model introduced in Section 3.1 and the Semantic Translation Operator introduced in Section 3.2.

Importantly, the hierarchy in Appendix A.1 classifies assurance targets rather than the expressive power of Lean itself. In principle, properties across all four levels may admit formalization in Lean under suitably enriched semantics. Under the current translation interface, however, the most directly supported fragment is the local transition fragment induced by $(\Sigma_{\mathcal{C}}, \Gamma, I_{\mathcal{C}}, \Upsilon_{\mathcal{C}})$ and its reified output. Temporal, ordering-sensitive, and fully strategic properties generally require enriched trace semantics or multi-agent outcome semantics beyond this default translation. At the same time, as the zap case study illustrates, once additional economic assumptions are explicitly reified into the formal model—such as typed asset flows, market state, token identity, and fair-value benchmarks—Lean can still certify snapshot-level economic guarantees within this interface.

A.2.1 Type-Theoretic Abstraction of State and Context

Formal verification requires a static definition of the system's state space. We model Solidity's storage layout and the relevant execution environment as explicit Lean structures.

The Immutable State Structure (Σ). Solidity's core feature is its slot-based mutable storage. To reason about contract evolution, we map the relevant storage layout into an immutable record type in Lean.

- **Scalar Mapping:** Each state variable v_i becomes a field in the structure Σ. Primitive types such as `uint256` are modeled according to the chosen arithmetic abstraction: `BitVec` 256 when modular EVM arithmetic must be preserved, and $\mathbb{N}$ when unbounded arithmetic suffices for the target proof obligations.

- **Functional Mappings:** A Solidity `mapping(K => V)` is abstracted as a total function space $K \to V$ together with Solidity's default-value semantics for unwritten keys. For example, a ledger can be modeled as Address $\to \mathbb{N}$. This yields Reference Transparency and enables point-wise reasoning via pure functional updates. When a property aggregates over all keys, an additional finite-support assumption must be stated explicitly.

Explicit Context Parameterization (Γ). Contract execution heavily depends on the blockchain's ambient context (e.g., `msg.sender`, `msg.value`, `block.timestamp`). In Solidity, these are implicit environmental variables.

- **Lifting:** We define an explicit structure Γ to encapsulate these environmental variables. In the Lean model, each contract function f becomes a pure function that takes Γ as an explicit argument.

- **Symbolic Determinism:** Once the environment is reified as an explicit parameter, execution is deterministic for any fixed $\gamma \in \Gamma$. This allows the prover to reason symbolically over all admissible environments by quantifying over Γ.

A.2.2 Reification of Side Effects and Monadic Output

Smart contracts produce side effects, primarily on-chain events and asset transfers. Since Lean is a pure functional language and does not execute EVM instructions directly, we reify these effects as data.

Transition Output (O). We represent a successful execution by an abstract output object

$$O := \{\, \text{state} : \Sigma,\ \text{events} : \text{List Event},\ \text{transfers} : \text{List Transfer} \,\}.$$

Its state component records the post-state $\sigma' \in \Sigma$, while the event and transfer fields record emitted logs and intended asset movements, respectively. This design allows side effects to be verified

logically. For example, proving "non-admins cannot withdraw" can be reduced to proving that the list `transfers` is empty under the relevant pre-conditions.

Inductive Result Type ($\mathcal{R}$). To capture the Atomicity of the EVM, we define

$$\mathcal{R} ::= \text{success}(out : O) \mid \text{failure}(err : \text{Error}).$$

A failure branch represents a reverted execution, whereas a success branch exposes the post-state and reified side effects. This structure forces the proof process to handle exceptional paths explicitly, eliminating verification blind spots caused by ignored `revert` branches.

A.2.3 The Guard-Action Modeling Paradigm

To bridge the semantic gap and ensure the provability of the generated Lean code, we adopt a standardized Guard-Action paradigm for each function-level transition. At the contract level, the aggregate transition function $\Upsilon_{\mathcal{C}}$ dispatches on the tagged input $\iota \in I_{\mathcal{C}}$ and applies the corresponding function-level rule. For a Solidity function f, we model its semantics as

$$\Upsilon_f(\sigma, \gamma, \iota) = \begin{cases} \text{failure}(e_f), & \neg P_f(\sigma, \gamma, \iota), \\ \text{success}(A_f(\sigma, \gamma, \iota)), & P_f(\sigma, \gamma, \iota), \end{cases}$$

where P_f is the guard predicate induced by `require`-style pre-conditions, and A_f constructs the reified output object.

Guard (Pre-condition Verification). Each `require` statement is compiled into an explicit guard. Inputs that do not satisfy the pre-conditions are mapped directly to `failure`, thereby pruning invalid execution branches from the proof space.

Action (Functional State Update). If the guard holds, the model enters the update phase. Mutable storage operations are simulated using pure functions:

- **Lambda Mapping Update.** For map updates such as `balances[user] = amount`, we model the new mapping by

  $$\text{new\_map} := \lambda a.\ \text{if } a = \text{user then amount else old\_map}(a).$$

  This precisely captures point-wise update semantics.

- **Record Update.** For ordinary storage fields, we use Lean's record update syntax to construct the new post-state.

- **Output Assembly.** The resulting post-state, emitted events, and transfer intents are assembled into the output object returned by `success`.

A.3 Translation Example

To make the abstract semantic translation concrete, we utilize a running example: a simple Auction Contract. This contract embodies common Solidity features such as mutable storage mappings, implicit context dependencies, conditional guards, and emitted events. Figure 4 provides a side-by-side comparison between the source Solidity logic and the generated Lean functional model.

A.3.1 Type-Theoretic Abstraction

Before translating the logic shown in Figure 4, we first map the relevant storage and implicit EVM context into static Lean structures.

The Immutable State. We translate the storage fields read or written by `bid` into a Lean structure. In particular, the Solidity variable `mapping(address => uint)` is modeled as a total function, enabling point-wise updates via lambda abstraction. Since the guard in Figure 4(a) also depends on `endTime`, it must be represented explicitly in the state. For expository simplicity, this example instantiates the numeric abstraction with `Nat`.

```
structure State where
  highestBidder  : Address
  highestBid     : Nat
  pendingReturns : Address -> Nat
  endTime        : Nat
```

The Explicit Context. Implicit EVM variables (e.g., `msg.sender`, `msg.value`, and `block.timestamp`) are reified into an explicit `Context` structure, making the external execution environment available as ordinary function arguments in Lean.

```
structure Context where
  sender    : Address
  value     : Nat
  timestamp : Nat
```

Consistent with Appendix A.2, a successful execution returns a reified output object whose fields record the post-state, emitted events, and transfer intents.

A.3.2 The Guard-Action Transformation

The core translation logic is visualized in Figure 4, and consists of two major steps.

From Reverts to Guards. In Solidity (Figure 4(a)), the `require` statements enforce preconditions by reverting the transaction. In the Lean model (Figure 4(b)), these are transformed into explicit conditional guards. If a condition fails (e.g., `ctx.value <= s.highestBid`), the function deterministically returns a `failure` result, pruning invalid execution paths from the proof space.

From Mutation and Side Effects to Functional Update and Reified Output. The most critical transformation occurs in the update phase. Solidity employs imperative mutation (Figure 4(a), Line 9: `pendingReturns[highestBidder] += highestBid`). In contrast, the Lean model uses functional actions. As shown in Figure 4(b), this is modeled as a *lambda update*: a new function is constructed that returns the updated balance for the previous highest bidder and the original balance for all other addresses. The post-state is then assembled with a record update. Finally, instead of executing side effects directly, the emitted event is reified into the returned output object. In this example, the transfer list is empty because `bid` records a pending refund rather than performing an immediate payout.

A.4 Adversarial Simulation Details

To capture economic attack vectors emerging from complex interactions (e.g., Price Manipulation, Sandwich Attacks), we complement the state-based verification interface of Section 3.3 with a dynamic adversarial simulation layer. This layer primarily targets properties whose semantics depend on histories, transaction ordering, or strategic multi-agent outcomes. We construct an LLM-driven sandbox where heterogeneous agents with long-term memory perform stress testing under a controlled evolutionary process.

A.5 Adversarial Simulation Details

To capture economic attack vectors emerging from complex interactions (e.g., Price Manipulation, Sandwich Attacks), we complement the local-transition verification interface of Section 3.3 with a dynamic adversarial simulation layer. This layer primarily targets properties whose semantics depend on histories, transaction ordering, or strategic multi-agent outcomes. We construct an LLM-driven sandbox where heterogeneous agents with long-term memory perform stress testing under a controlled evolutionary process.

A.5.1 Simulation Environment

We define the simulation environment as a tuple

$$\mathcal{E} = \langle \mathcal{S}, \mathcal{TX}, \mathcal{P}, \Omega \rangle.$$

$\mathcal{S}$: The Global Blockchain State Space. It contains the target contract's storage (Σ defined in Appendix A.2) as a distinguished component, and augments it with dynamically integrated external dependencies (e.g., DEX liquidity pools, lending protocols) to ensure high-fidelity ecological simulation.

$\mathcal{TX}$: The set of admissible transactions, representing the action space available to agents.

$\mathcal{P} : \mathcal{S} \times \mathcal{TX} \to \mathcal{S}$: The state transition function induced by the simulator.

Ω: The observation function, through which agents acquire public on-chain data as local observations o_t.

Because the simulator enriches Σ with external mutable dependencies, it should be viewed as an empirical extension of the closed-system formal semantics of Appendix A.2, rather than a replacement for the Lean model.

We additionally maintain a finite execution history

$$h_t := \langle s_0, \tau_0, s_1, \ldots, \tau_{t-1}, s_t \rangle,$$

which records the causal order of actions and is required for temporal and ordering-sensitive properties.

A.5.2 The Runtime Property Oracle ($\mathcal{O}$)

To strictly enforce the formal specifications used by the Verifier in Section 4.2 within the dynamic sandbox, we implement a Property Oracle, denoted as a deterministic monitor function

$$\mathcal{O} : \mathcal{H} \times \hat{\Phi}(R) \to \{\top, \bot\},$$

where $\mathcal{H}$ is the set of finite execution histories induced by $\mathcal{P}$.

Operationally, we decompose the monitored property set into

$$\hat{\Phi}(R) = \hat{\Phi}_{\text{local}}(R) \cup \hat{\Phi}_{\text{trace}}(R) \cup \hat{\Phi}_{\text{game}}(R).$$

For a current history h_t ending in state s_t, the oracle evaluates local- and trace-level specifications by

$$\mathcal{O}(h_t, \hat{\Phi}(R)) \Leftrightarrow \left(\bigwedge_{\phi \in \hat{\Phi}_{\text{local}}(R)} \phi(s_t) \right) \wedge \left(\bigwedge_{\psi \in \hat{\Phi}_{\text{trace}}(R)} \psi(h_t) \right),$$

```solidity
function bid() public payable {
    // 1. Pre-condition Checks
    require(block.timestamp < endTime, "Ended");
    require(msg.value > highestBid, "Too low");

    // 2. Mutable State Update
    if (highestBid != 0) {
        pendingReturns[highestBidder] += highestBid;
    }

    // 3. Slot Updates & Side Effects
    highestBidder = msg.sender;
    highestBid = msg.value;
    emit HighestBidIncreased(msg.sender, msg.value);
}
```

(a) Source: Imperative Solidity Logic

```
def bid (ctx : Context) (s : State) : Result :=
  -- 1. Conditional Guards (Reified Reverts)
  if ctx.timestamp >= s.endTime then
    failure Ended
  else if ctx.value <= s.highestBid then
    failure Low
  else
    -- 2. Functional Action (Lambda Update)
    let pendingReturns' :=
      if s.highestBid != 0 then
        fun a =>
          if a = s.highestBidder then
            s.pendingReturns a + s.highestBid
          else
            s.pendingReturns a
      else
        s.pendingReturns

    -- 3. New State Construction
    let s' := { s with
      highestBidder  := ctx.sender
      highestBid     := ctx.value
      pendingReturns := pendingReturns'
    }

    -- 4. Reified Output
    success {
      state     := s',
      events    := [Event.HighestBidIncreased
     ctx.sender ctx.value],
      transfers := []
    }
```

(b) Target: Functional Lean Model

Figure 4: **The Semantic Translation of the `bid` Function.** The imperative Solidity logic in (a) is transformed into a pure Lean transition in (b). Note the correspondence between `require` statements and conditional guards, the modeling of mapping mutation as a lambda update, and the reification of side effects into the returned output object.

and, at designated checkpoints (typically episode boundaries or task-specific horizons), it additionally evaluates game-level specifications

$$\bigwedge_{\chi \in \hat{\Phi}_{\text{game}}(R)} \chi(\text{outcome}(h_t)),$$

where $\text{outcome}(h_t)$ summarizes utilities, allocations, or other payoff-relevant observables derived from the episode.

Unlike heuristic-based fuzzers that only look for crashes, $\mathcal{O}$ serves as a semantic bridge: local properties are checked on the current simulator state, temporal and ordering properties are checked on histories, and incentive-oriented properties are checked on aggregated outcomes.

If $\mathcal{O}(h_t, \hat{\Phi}(R))$ evaluates to $\perp$ (False), the simulation is immediately halted. The sequence of actions leading to the violation is returned as a Counter-example Trace

$$\tau_{\text{seq}} := h_t,$$

providing the exact causal chain for the feedback synthesis module. When the resulting pattern can be expressed as a local- or trace-level constraint, the AttackAnalyst distills it into a new property and injects it back into $\hat{\Phi}(R)$ for subsequent refinement and verification.

A.5.3 The Cognitive Architecture

The participant set in the POMG is modeled as

$$\mathbf{A} = \{A_{honest}, A_{attacker}\}.$$

Each agent A_i consists of an LLM-driven policy π_θ and a dynamic memory module $\mathcal{M}_i$.

Unlike stateless fuzzing, our agents possess temporal cognition. At timestep t, an agent receives observation o_t and recalls history $\mathcal{M}_{t-1}$ to synthesize a concrete Solidity execution script

$$\tau_t \sim \pi_\theta(o_t, \mathcal{M}_{t-1}).$$

For Honest Agents A_{honest}, the policy is constrained to follow intended protocol behaviors (e.g., providing liquidity), thereby simulating organic market noise. For Adversarial Agents $A_{attacker}$, the objective is to maximize personal utility or violate monitored properties. The memory $\mathcal{M}_i$ records feedback from past attempts (e.g., revert reasons, gas costs, and prior near-misses), enabling

multi-turn trial and error and progressively more effective exploit strategies.

A.5.4 Adversarial Interleaved Scheduling

To realistically simulate Mempool disorder and MEV (Miner Extractable Value) risks, we employ an Adversarial Interleaved Scheduling strategy. Instead of a fixed FIFO order, the attacker is granted the privilege to observe pending honest transactions and intervene.

Specifically, when an honest transaction τ_{honest} is submitted, the system suspends it and exposes the intent to the attacker. The attacker can then insert a malicious transaction τ_{attack} at a chosen position. This mechanism explores the combinatorial space of transaction ordering, enabling: Front-running (executing $\langle \tau_{attack}, \tau_{honest} \rangle$ to preempt state changes) and Sandwiching (executing $\langle \tau_{front}, \tau_{honest}, \tau_{back} \rangle$ to manipulate prices).

The complete loop, from environment initialization to adversarial scheduling and property monitoring, is detailed in Algorithm 1.

B Case Study: Repairing a Curve-Style Zap-In Contract

This appendix presents the case study in the same order as the LeVer loop in Figure 2 of the main paper: a *Coder* first proposes a Solidity candidate C_k, the *Verifier* translates it into Lean and attempts proofs, the *Attacker* produces a concrete sandbox trace, and the *AttackAnalyst/Analyst* converts that trace into structured feedback F_k for the next round. All stages in this pipeline are instantiated using Gemini 3 Pro as the underlying foundation model. To keep the presentation compact enough to show the full iterative workflow, the Lean code displayed in this case study consists of representative excerpts rather than the full artifact, and some intermediate theorem statements are shown with `sorry` placeholders to illustrate how obligations are introduced during the loop. The complete final Lean artifact for this case study, including the finished proof scripts and detailed explanation, is provided in Appendix D. The emphasis here is therefore not on the contract as an isolated artifact, but on how one contract is progressively repaired as the executable code, the formal model, and the property set coevolve.

Task specification

This smart contract is a [Zap-in transaction contract] for liquidity provision on Curve Finance. The main functions include [approving ERC20 tokens, processing ERC20-based zap-in transactions, checking the active or stopped status of the contract, executing swaps or liquidity-entry computations, enforcing a user-provided minimum liquidity bound, and returning the received Curve pool tokens to the caller]. The contract transitions between states based on user interactions: [First, users call the ZapIn or ZapInWithERC20 function with a specified input token, input amount, and minimum acceptable liquidity. The contract checks the relevant approval and token conditions, pulls the input ERC20 asset from the user, computes the amount of pool tokens according to the current market or pool state, verifies that the output satisfies the minLiquidity constraint, and then transfers the received pool tokens back to the user]. The main variables include [goodwill, stopped, exchange2Token, supported token addresses, pool token addresses, token allowances, market reserves, and price or precision parameters used for slippage calculation]. The main functions include [approveToken, ZapIn, ZapInWithERC20, quoteMinLiquidity, swap or liquidity-entry helper functions, and token validation functions]. The main events include [contract activation or pausing, token approval confirmations, and zap-in execution confirmations].

B.1 Background

A *zap-in* contract is a wrapper that lets a user deposit one asset and receive a liquidity-provider (LP) token representing a position in a pool. In Ethereum-like systems, fungible assets usually follow the ERC-20 interface: a user first grants permission through `approve`, and a protocol later pulls funds via `transferFrom`. This is important because a protocol may maintain its own internal allowance bookkeeping, but that bookkeeping has no financial effect unless a real token transfer is executed. A zap-in wrapper is therefore secure only if three questions are answered correctly: who actually pays, how output is priced, and how the lower-bound slippage guard is calibrated.

Two further notions are needed for readers outside blockchain. First, an *automated market maker* (AMM) determines swap output from pool reserves rather than from an order book. Second, many DeFi interfaces expose a user-specified lower bound such as `minLiquidity`. The transaction succeeds only if realized output is at least this bound. In practice, the bound is often expressed through a *slippage tolerance*, frequently measured in *basis points* (bps), where 100 bps equals 1%. This becomes subtle when the input and output tokens use different decimal systems or represent different assets: an input amount denominated in token A cannot be directly compared to an output amount

Algorithm 1 Adversarial Agent-Based Sandbox Simulation

Require: Contract Code C, Property Set $\hat{\Phi}(R)$, Max Rounds $R_{\max}$
Ensure: Safety Report or Attack Trace τ_{seq}

Initialize simulation environment $\mathcal{E}_0$ with C and mocks
Initialize history $h \leftarrow \langle s_0 \rangle$
Initialize agents $\mathcal{A} = \{A^1_{honest}, \ldots, A^N_{honest}, A^1_{attacker}, \ldots, A^M_{attacker}\}$
Initialize memory $\mathcal{M}_i \leftarrow \emptyset$ for all $A_i \in \mathcal{A}$
for $r = 1$ to $R_{\max}$ **do**
    $\mathcal{Q} \leftarrow \text{SHUFFLE}(A_{honest})$ ▷ Randomize honest user execution order
    for $A_h \in \mathcal{Q}$ **do**
        $o_h \leftarrow \text{OBSERVE}(\mathcal{E}_{r,\text{step}})$
        $\tau_h \leftarrow A_h.\text{ACT}(o_h, \mathcal{M}_h)$ ▷ Generate honest transaction intent
        ▷ **Adversarial Interleaving Step**
        for $A_m \in A_{attacker}$ **do**
            $o_m \leftarrow \text{OBSERVE}(\mathcal{E}_{r,\text{step}}) \cup \{\tau_h\}$ ▷ Attacker sees state and honest tx
            $\tau_m \leftarrow A_m.\text{ACT}(o_m, \mathcal{M}_m)$
            if $\tau_m \neq \emptyset$ **then**
                Apply τ_m to $\mathcal{E}$
                Update history h with the new transition
                if $\mathcal{O}(h, \hat{\Phi}(R)) = \bot$ **then**
                    return Found Vulnerability h
                end if
                Update $\mathcal{M}_m$ based on execution feedback
            end if
        end for
        ▷ **Execute Honest Transaction**
        Apply τ_h to $\mathcal{E}$
        Update history h with the new transition
        if $\mathcal{O}(h, \hat{\Phi}(R)) = \bot$ **then**
            return Found Vulnerability h
        end if
        Update $\mathcal{M}_h$ based on execution feedback
    end for
end for
return Safe (No Violation Found)

denominated in token B.

Artifacts shown in each round

Each round contains four aligned artifacts: (i) the current Solidity candidate, (ii) a representative Lean excerpt, (iii) the attacker trace that still succeeds or now fails, and (iv) the analyst feedback that defines the next repair target. The Lean snippets are intentionally selective: they show the formal delta that matters for the discovered bug, while baseline guard, atomicity, and regression properties are generated and checked throughout the loop.

B.2 Round 0 : Missing Payment Semantics

Round 0: Initial candidate C_0
The first contract captures the broad zap-in control flow, but it only records internal approval bookkeeping and never actually pulls the caller's tokens.

As Figures 5 and 6 show, the first-round candidate captures the broad zap-in control flow but still lacks a coupled payment semantics: the Solidity code records only internal approval bookkeeping and an outgoing pool-token transfer, while the Lean model can describe payouts but not a matched user-to-contract payment. The deeper problem is

```
mapping(address => mapping(address => uint256
    ))
    public tokenAllowance;

function approveToken(address token, uint256
    amount) external {
    require(token != address(0), "
    InvalidTokenAddress");
    tokenAllowance[msg.sender][token] =
    amount;
    emit Approved(msg.sender, token, amount);
}

function simulateSwap(address /* inToken */,
    uint256 inAmount)
    public pure returns (uint256)
{
    require(inAmount > 0, "InsufficientInput"
    );
    return (inAmount * 99) / 100;
}

function zapIn(address tokenIn, uint256
    amountIn,
                uint256 minLiquidity) external
     {
    require(tokenAllowance[msg.sender][
    tokenIn] >= amountIn,
            "InsufficientAllowance");
    uint256 swapAmount = simulateSwap(tokenIn,
     amountIn);
    uint256 poolTokens = (swapAmount * 9) /
    10;
    require(poolTokens >= minLiquidity,
            "SlippageToleranceExceeded");
    // BUG: no transferFrom(msg.sender,
    address(this), amountIn)
    IERC20(poolToken).transfer(msg.sender,
    poolTokens);
}
```

Solidity C_0: protocol-side bookkeeping + static stub

Figure 5: **Round 0 (Solidity).** The initial candidate records internal approval bookkeeping but does not pull the caller's tokens.

semantic: the successful path has an outgoing LP transfer, but neither the executable contract nor the formal side-effect vocabulary yet forces a matching incoming transfer from the user. The Verifier can still check basic guard and atomicity properties in this round; what it cannot state precisely is the newly needed fair-exchange relation, because the model can describe "the user receives an amount" but not "the user paid token X and then received token Y."

```
structure TransitionOutput where
  new_state : State
  events    : List Event
  transfers : List (Address x Amount)

def simulate_swap (_inToken : Address)
    (inAmount : Amount) (_s : State) : Option
     Amount :=
  if inAmount > 0 then
    some (inAmount * 99 / 100)
  else
    none

let transfers : List (Address x Amount) :=
  [(ctx.sender, poolTokens)]

CallResult.success {
  new_state := newState,
  events := [Event.ZapInExecuted ctx.sender
              inAmount poolTokens],
  transfers := transfers
}

-- The full round also includes baseline guard and
-- atomicity obligations. The excerpt shows the
     missing
-- vocabulary: no payer field and no token
     identity.
```

Lean V1: payout abstraction

Figure 6: **Round 0 (Lean).** The initial model can express payouts, but not a matched payment from the user.

Attacker witness — τ_0

```
=== Aligned V1 Faucet Trace ===
phantom amountIn: 100000000000000000000
internal allowance before:
     100000000000000000000
internal allowance after:
     100000000000000000000
fake token balance before: 0
fake token balance after:  0
LP gained:
     891000000000000000
USDC before:               0
redeemed USDC:
     891000000000000000

USDC after:
     891000000000000000
```

The trace shows the vulnerability in the most direct possible way. The attacker first writes an internal allowance entry for a fake token, then calls `zapIn` with `amountIn=100`. The allowance remains unchanged, the fake token balance stays at zero, yet the caller still receives 89.1 LP and then redeems that LP for 89.1 USDC. This is exactly the behavior of a *faucet*: the contract dispenses value without enforcing payment. The sandbox witness therefore tells the Verifier what kind of property is still missing.

AttackAnalyst feedback — F_0

The next round must repair both layers simultaneously. On the Solidity side, the contract

needs an actual user-to-contract payment path through `transferFrom`. On the Lean side, the output representation must be enriched so the model can explicitly state who pays whom, in which token, and in what amount.

B.3 Round 1 : Static Pricing

Round 1: Candidate C_1

The second contract adds a real pull of the input asset and consumes allowance correctly, but it still prices the trade through a fixed stub instead of an explicit market state.

As Figures 7 and 8 show, this revision is more than a local bug fix. The Solidity candidate now performs a real `transferFrom` and consumes allowance, while the Lean model upgrades the side-effect representation from anonymous payout tuples to typed `TransferActions`, making the new payment and allowance obligations explicit.The Solidity change ensures that successful execution now consumes a real ERC-20 payment, and the Lean change upgrades the side-effect language from anonymous payout tuples to typed transfer actions. As a consequence, the proof layer can now formulate statements such as `zap_in_enforces_payment` and `zap_in_reduces_allowance`. The displayed `sorry` placeholders are the newly induced obligations from the previous attack trace; they are not the entire verification workload. Previously generated guard and slippage obligations continue to be checked as regression properties, while the important new capability is that fair exchange has become formally stateable.

Attacker witness — τ_1

```
=== Aligned V2 Static-Price Trace ===
amountIn:
     100000000000000000000
internal allowance before:
     100000000000000000000
internal allowance after:  0
USDC before:
     100000000000000000000
USDC after zap:            0
LP received:
     89100000000000000000
redeemed USDC:
     178200000000000000000
final USDC:
     178200000000000000000
profit:
     78200000000000000000
```

The faucet bug is gone: the attacker really spends 100 USDC and the allowance drops to zero. But the sandbox still breaks the economic intent. Because output is derived from a fixed pricing stub, the attacker receives 89.1 LP, redeems that LP elsewhere for 178.2 USDC, and realizes a profit of 78.2 USDC. In other words, payment semantics are now correct, but pricing semantics are still unrealistic.

```solidity
function zapIn(address tokenIn, uint256 amountIn,
               uint256 minLiquidity) external {
    require(tokenAllowance[msg.sender][tokenIn] >=
      amountIn,
            "InsufficientAllowance");

    bool pulled = IERC20(tokenIn).transferFrom(
        msg.sender, address(this), amountIn
    );
    require(pulled, "PullFailed");

    uint256 swapAmount = simulateSwap(tokenIn,
     amountIn);
    uint256 poolTokens = (swapAmount * 9) / 10;
    require(poolTokens >= minLiquidity,
            "SlippageToleranceExceeded");

    unchecked { tokenAllowance[msg.sender][tokenIn
     ] -= amountIn; }
    IERC20(poolToken).transfer(msg.sender,
     poolTokens);
}
```

Solidity C_1: real pull + allowance consumption

Figure 7: **Round 1 (Solidity).** The candidate now pulls the user's asset and consumes the recorded allowance.

```
structure TransferAction where
  fromAddr : Address
  toAddr   : Address
  token    : Address
  amount   : Amount

structure TransitionOutput_Fixed where
  new_state : State
  events    : List Event
  transfers : List TransferAction

-- Newly induced by F0:
theorem zap_in_enforces_payment ... := by sorry
theorem zap_in_reduces_allowance ... := by sorry

-- Prior guard/slippage obligations are retained.
theorem zap_in_guarantees_slippage ... := by sorry
```

Lean V2: transfer actions + induced obligations

Figure 8: **Round 1 (Lean).** The model now represents typed asset transfers and the new payment-related obligations.

AttackAnalyst feedback — F_1

The next round cannot continue to treat the zap as a fixed-percentage dispenser. The Coder must replace the static quote stub with a market-aware mechanism, and the Verifier must lift that change into an explicit market model whose reserves become part of the transition context.

```
MockAMMPool public immutable market;

function zapIn(address tokenIn, uint256
    amountIn,
                uint256 minLiquidity) external
     {
    require(tokenAllowance[msg.sender][
     tokenIn] >= amountIn,
            "InsufficientAllowance");

    bool pulled = IERC20(tokenIn).
     transferFrom(
        msg.sender, address(this), amountIn
    );
    require(pulled, "PullFailed");

    uint256 actualOut = market.get_dy(
     amountIn);
    require(actualOut >= minLiquidity,
            "SlippageToleranceExceeded");
    unchecked { tokenAllowance[msg.sender][
     tokenIn] -= amountIn; }
    IERC20(poolToken).transfer(msg.sender,
     actualOut);
}
```

Solidity C_2: explicit market object

Figure 9: **Round 2 (Solidity).** The candidate replaces the fixed stub with an explicit AMM object for market-aware pricing.

B.4 Round 2 : Market-Aware but Token-Unsafe

Round 2: Candidate C_2

The third contract stops using a fixed output stub and instead queries an AMM-like market object. This repairs the pricing abstraction but still leaves token identity unchecked.

As Figures 9 and 10 show, the contract is no longer a fixed-price dispenser. The executable candidate now queries an explicit AMM object, and the Lean model introduces a corresponding `MarketState` together with a market-aware bounded-loss theorem. The executable code queries an AMM object, and the Lean model introduces a `MarketState` whose reserves determine the output function. This is the round where a newly induced economic obligation is discharged: the completed theorem establishes a bounded-loss property over an explicit market snapshot. This does not mean proof search starts only here; rather, the proof set now includes a harder market-aware property in addition to the carried-forward execution and payment properties. However, one key assumption remains unmodeled: the contract still treats the caller-provided `tokenIn` as if any asset were a valid input to the market.

```
structure MarketState where
  reserveIn  : Amount
  reserveOut : Amount

structure Context where
  sender : Address
  market : MarketState

def get_amm_out (market : MarketState) (dx :
     Amount) : Amount :=
  let x := market.reserveIn
  let y := market.reserveOut
  if x + dx == 0 then 0 else
    (y * dx) / (x + dx)

theorem slippage_mechanism_is_robust ... :=
     by
  ...
```

Lean V3: market state + bounded-loss proof

Figure 10: **Round 2 (Lean).** The model introduces explicit market state and the bounded-loss theorem.

Attacker witness — τ_2

```
=== Aligned V3 Fake-Token Trace ===
fake token spent:
     100000000000000000000
internal allowance before:
     100000000000000000000
internal allowance after:  0
LP received:
     500000000000000000000
redeemed USDC:
     500000000000000000000
USDC before:               0
USDC after:
     500000000000000000000
```

This trace is especially revealing because it shows that *quantity correctness* is not the same as *asset correctness*. The attacker really does spend 100 FAKE tokens, and the internal allowance is correctly decremented. But because the AMM formula is called without validating that `tokenIn` matches the market's accepted input asset, the contract still returns 500 LP, which can then be redeemed for 500 USDC. The pricing mechanism is now market-aware, yet the market is effectively being queried on behalf of the wrong asset.

AttackAnalyst feedback — F_2

Two new repair targets emerge. First, successful execution must require the user-supplied token to equal the market's accepted input token. Second, the slippage baseline must be recalibrated: it is not enough to compare `actualOut` with a threshold derived directly from `amountIn`, because input and output tokens may have different decimals and different market meanings.

```
uint256 public fairPrice;
uint256 public pricePrecision;

function calcExpectedOut(uint256 amountIn)
    public view returns (uint256)
{
    if (pricePrecision == 0) return 0;
    return (amountIn * fairPrice) /
     pricePrecision;
}

function quoteMinLiquidity(uint256 amountIn,
     uint256 toleranceBps)
    public view returns (uint256)
{
    uint256 expectedOut = calcExpectedOut(
     amountIn);
    return constructSafeMinLiquidity(
     expectedOut, toleranceBps);
}

function zapIn(address tokenIn, uint256
     amountIn,
               uint256 minLiquidity) external
      {
    require(tokenIn == address(market.tokenIn
     ()),
            "InvalidTokenAddress");
    uint256 actualOut = market.get_dy(
     amountIn);
    require(actualOut >= minLiquidity,
            "SlippageToleranceExceeded");
}
```

Solidity C_3: token check + fair-value helpers

Figure 11: **Round 3 (Solidity).** The final candidate adds a token-identity check and fair-value-based helper functions for computing the slippage threshold.

B.5 Round 3 : Token Validation and Fair-Value Slippage

Round 3: Candidate C_3
The fourth contract blocks the fake-token path and replaces the raw amount-based slippage floor with a fair-value-based baseline that is dimensionally meaningful.

As Figures 11 and 12 show, the final round fixes two conceptually different problems. The Solidity candidate adds token-identity validation and fair-value helper functions, while the Lean model strengthens the market semantics accordingly and states the final slippage theorem over the refined token-aware setting.The first is token validity: the candidate now rejects any `tokenIn` that does not match the market's accepted input asset. The second is a dimension mismatch. In heterogeneous-decimal settings such as WBTC/WETH, an input quantity like `100000000` (which can mean 1 WBTC with 8 decimals) cannot serve as a meaningful lower bound for an output amount measured in a 18-decimal token. The repaired design therefore computes a fair-value benchmark `expectedOut`, then derives `minLiquidity` from that benchmark rather than from the raw input amount.

```
structure MarketState where
  tokenIn    : Address
  tokenOut   : Address
  reserveIn  : Amount
  reserveOut : Amount

structure Context where
  sender         : Address
  market         : MarketState
  fairPrice      : Amount
  pricePrecision : Amount

theorem slippage_mechanism_is_robust
  ...
  (h_token_valid : inToken = market.tokenIn) :
  let expectedOut := calc_expected_out amountIn
     fairPrice
                       pricePrecision
  let actualOut := get_amm_out market amountIn
  calc_loss_bps expectedOut actualOut <=
     toleranceBps := by
  ...

-- Prior asset-flow and market-state properties
     are
-- rechecked under the stronger token-aware model.
```

Lean V4: token-aware market + stronger theorem

Figure 12: **Round 3 (Lean).** The final model is token-aware and proves the stronger slippage theorem under the refined market semantics.

Attacker witness — τ_3

```
=== Aligned V4 Security Trace ===
fair expected out:
     14000000000000000000
spot output from AMM:
     13861386138613861386
wrong amountIn-based minLiquidity: 99000000
correct fair-value-based minLiquidity:

     13860000000000000000
actual out:
     13861386138613861386
loss vs fair benchmark (bps):      99
```

The first half of the security witness is a failed exploit: a fake-token call now reverts with `InvalidTokenAddress`. The second half is more subtle and more important. It shows why the earlier amount-based slippage floor is meaningless. The old formula would have accepted a threshold of only $99,000,000$, which is tiny in the output token's unit system. The repaired formula instead yields a lower bound near 13.86×10^{18}, which is commensurate with the output asset. The realized output remains within 99 bps of the fair-value benchmark, so the benign trade succeeds for the right reason.

AttackAnalyst feedback — F_2

At this point the main discovered attack families are blocked under the explicit modeling assumptions: there is no free LP minting, no fixed-stub arbitrage, no fake-token path, and no dimensionally meaningless slippage floor. The appendix should still state the boundary clearly: the final Lean theorem is a proof over a deterministic closed-system model with an explicit market snapshot and an explicit fair benchmark. It is not, by itself, a proof of full real-world MEV resistance in the live mempool.

B.6 Compact Summary Across Rounds

Table 2 summarizes the four rounds of the zap-in case study, highlighting how each attacker trace changes both the next Solidity candidate and the next formal property set.

B.7 Takeaway

Presented in the same order as the framework itself and summarized in Table 2, the four rounds make the LeVer loop concrete. The Coder does not merely patch syntax: each iteration changes what the contract is allowed to mean. The Verifier does not merely check finished code: in every round it maintains and rechecks the current property set while progressively strengthening the formal object, moving from anonymous payouts to typed asset flows, then to explicit market state, and finally to token-aware and dimension-consistent economic guarantees. The Attacker is not redundant with the Verifier because the trace reveals precisely which semantic assumption is still too weak. The AttackAnalyst is therefore the crucial bridge: it transforms a concrete exploit witness into the next property that the system must be able to state and eventually prove.

C Auction Case Study: Repairing the Closing Path of an English Auction

Task specification

This smart contract is an [English auction contract] for selling an NFT in exchange for ETH. The main functions include [escrowing the NFT, accepting bids before the auction deadline, tracking the highest bidder and highest bid, recording refundable balances for outbid users, finalizing the auction, transferring the NFT to the winner, and making the winning payment available to the seller]. The contract transitions between states based on user interactions: [First, users call the bid function with ETH before the auction ends. A valid bid must be higher than the current highestBid, the previous highest bidder's funds are recorded as withdrawable credit, and highestBidder and highestBid are updated. After the deadline, users call endAuction to close the auction, mark the auction as ended, transfer the NFT to the highest bidder, and record the seller's claim to the winning payment without blocking finalization on the seller's ability to receive ETH. Users with pending credits call withdraw to retrieve their funds]. The main variables include [seller, nft, tokenId, endTime, ended, highestBidder, highestBid, and pendingReturns]. The main functions include [bid, endAuction, and withdraw]. The main events include [new highest bid notifications, auction finalization confirmations, and withdrawal confirmations].

C.1 Background

An *English auction* contract escrows an NFT, accepts bids before a deadline, and eventually transfers the NFT to the winner while making the winning payment available to the seller. In Ethereum-like systems, this closing path becomes subtle because ETH transfers are not guaranteed to succeed: a seller may be an externally owned account, but may also be a contract that rejects ETH in its `receive` or `fallback` function. A secure closing path must therefore answer two questions correctly: whether auction finalization is blocked by seller-side runtime behavior, and whether the contract's internal accounting remains solvent after closure.

A second notion is the distinction between *push* and *pull* payments. In a push-payment design, the contract attempts to send ETH to the seller during `endAuction`. In a pull-payment design, `endAuction` only records a seller credit, and the seller later calls `withdraw()` to retrieve the funds. This distinction matters because the former can couple auction liveness to seller behavior, while the latter introduces new accounting obligations over credits and residual state.

Artifacts shown in each round

Each round contains four aligned artifacts: (i) the current Solidity candidate, (ii) a representative Lean excerpt, (iii) the attacker trace that still succeeds or now fails, and (iv) the analyst feedback that defines the next repair target. The Lean snippets are intentionally selective: they show the formal delta that matters for the discovered bug, while baseline guard, closure, and regression properties are generated and checked throughout the loop.

Round	Coder output	Verifier delta	Attacker witness	Analyst feedback
V1 / C_0	Internal allowance bookkeeping and a static pricing stub; no real pull of the user's token.	Baseline guard/atomicity checks are generated; the excerpt exposes that anonymous payout tuples cannot express fair exchange.	The attacker pays 0, receives 89.1 LP, and redeems it for 89.1 USDC.	Introduce an actual payment path and enrich the formal side-effect language so payment and payout can be coupled.
V2 / C_1	Adds `transferFrom` and consumes allowance correctly.	Typed `TransferActions` appear; newly induced payment/allowance obligations are added while prior checks are retained.	The attacker spends 100 USDC but still profits because output is derived from a fixed stub.	Replace static pricing with an explicit market-state model.
V3 / C_2	Output depends on AMM reserves via `market.get_dy`.	`MarketState` is added; the new bounded-loss theorem is completed and prior payment obligations are rechecked.	The attacker spends 100 FAKE, receives 500 LP, and redeems 500 USDC.	Require token validity and redefine the slippage baseline in compatible units.
V4 / C_3	Adds a token-identity check and fair-value-based `minLiquidity` helpers.	The market includes token identities; the final theorem uses `expectedOut`, with prior obligations rechecked under the stronger model.	Fake-token path reverts; benign trade succeeds with a measured loss of 99 bps.	Main discovered attack families are blocked under the chosen deterministic market model.

Table 2: **One row per iteration of the LeVer loop in the zap-in case study.**

```
function endAuction() external {
    if (ended) revert AuctionAlreadyEnded();
    if (block.timestamp < endTime)
        revert AuctionNotYetEnded();

    ended = true;

    // Push ETH to the seller during
     finalization.
    (bool success, ) = payable(seller).call{
        value: highestBid
    }("");
    if (!success) revert TransferFailed();

    nft.transferFrom(address(this),
     highestBidder, tokenId);
    emit AuctionEnded(highestBidder,
     highestBid);
}
```

Solidity C_0: push-mode closing

Figure 13: **Round 0 (Solidity).** The initial candidate finalizes the auction by pushing ETH directly to the seller during `endAuction`.

C.2 Round 0 : Blocking Finalization

Round 0: Initial candidate C_0

The first contract captures the expected English-auction closing flow, but finalization still depends on the seller's ability to accept ETH in the same transaction.

Verifier status. As shown in Figure 13 and Figure 14, the initial Lean model already captures an important environmental dependency: whether the seller accepts ETH is not purely a property of the contract state, but of the execution context. The excerpt highlights the dependency that later becomes security-critical; it should not be read as saying that the first round only builds datatypes or avoids proofs. Basic guard and closure obligations are part of the verifier's workload from the beginning, while the attacker reveals that the property set still lacks non-blocking finalization.

The key issue is not merely a problematic ETH transfer primitive. The deeper problem is semantic: the closing path couples auction finalization to seller-side runtime behavior. In this round, the Lean model already captures that dependency through the environmental bit `sellerAcceptsEther`. The Verifier can therefore check baseline guard and closure obligations, but the attacker reveals that the current property set still lacks the stronger non-blocking-finalization requirement.

Attacker witness — τ_0

```
=== Auction V1 Push-Mode DoS Trace ===
highest bid: 1000000000000000000
auction balance after failed end:
     1000000000000000000
nft owner after failed end: 0x...A820a
ended after failed end: 0
```

```
def endAuction_v1 (ctx : Context) (s : State)
  : CallResultV1 :=
  if s.ended then .failure .
     AuctionAlreadyEnded
  else if ctx.timestamp < s.endTime then
    .failure .AuctionNotYetEnded
  else if s.highestBid > 0 &&
          (!ctx.sellerAcceptsEther) then
    .failure .TransferFailed
  else
    let newState := { s with
      ended := true
      contractBalance :=
        s.contractBalance - s.highestBid
      nftOwner := s.highestBidder }
    .success {
      newState := newState,
      events :=
        [Event.AuctionEnded s.highestBidder s.
     highestBid],
      transfers := [(s.seller, s.highestBid)]
      }

-- Baseline closure obligations are generated
     here.
-- The failed property discovered by the
     attacker is:
-- finalization should not depend on
     sellerAcceptsEther.
```

Lean V1 excerpt: push dependency and closure obligations

Figure 14: **Round 0 (Lean).** The model makes the push-payment dependency explicit through the environmental condition `sellerAcceptsEther`.

The sandbox deploys a `RevertingSeller` contract as the seller. Once the bidder has placed a valid winning bid and the deadline passes, `endAuction()` reverts with `TransferFailed`. Because Ethereum transactions are atomic, the revert rolls back `ended = true` as well. The auction therefore remains open in storage while being practically unfinishable: the NFT stays escrowed inside the auction contract, and the winning ETH remains locked there as well. The witness tells the Verifier exactly what is still missing: finalization should not depend on seller-side fund reception.

AttackAnalyst feedback — F_0

The next round must separate state finalization from seller-side ETH reception. On the Solidity side, the contract should adopt the pull-payment pattern: `endAuction` should record a seller credit and transfer the NFT to the winner, while the seller later retrieves the ETH via `withdraw()`. On the Lean side, the next property target is non-blocking closure with respect to seller behavior.

```
function endAuction() external {
    if (ended) revert AuctionAlreadyEnded();
    if (block.timestamp < endTime)
        revert AuctionNotYetEnded();

    ended = true;

    if (highestBid > 0) {
        pendingReturns[seller] += highestBid;
        // BUG: missing highestBid = 0;
    }

    nft.transferFrom(address(this), highestBidder,
      tokenId);
    emit AuctionEnded(highestBidder, highestBid);
}
```

Solidity C_1: pull pattern with stale `highestBid`

Figure 15: **Round 1 (Solidity).** The revised closing path uses pull payments, but leaves `highestBid` stale after crediting the seller.

C.3 Round 1 : Pull Payments but Stale Debt

Round 1: Candidate C_1

The second contract adopts pull payments and removes seller-dependent blocking at closure, but it leaves the winning-bid field stale after crediting the seller.

This revision is more than a local refactor (Figure 15 and Figure 16). The Solidity change removes the seller-dependent external call from `endAuction`, so the push-mode denial-of-service trace no longer applies. But once the interaction-level liveness issue is repaired, the next bottleneck shifts to internal accounting: does the contract's recorded liability still match the ETH it actually holds after closure? In this reduced witness, the relevant balance-sheet quantity is `pendingReturns[seller] + highestBid`.

Attacker witness — τ_1

```
=== Auction V2 Double-Accounting Trace ===
seller pending after end: 10000000000000000000
stale highestBid after end:
     10000000000000000000
contract balance: 10000000000000000000
recorded liability: 20000000000000000000
insolvency gap: 10000000000000000000
```

The attack here is not a profit-making exploit but an invariant-breaking witness. The NFT is correctly transferred to the winner, and the seller is credited once in `pendingReturns`. Yet `highestBid` still stores the same 10 ETH as if it remained live debt. The contract therefore reports liabilities of 20 ETH while actually holding only 10 ETH. The trace shows that pull payments alone are not enough: the

```
def totalLiability (s : State) : Amount :=
  s.pendingReturns s.seller + s.highestBid

def endAuction_v2 (ctx : Context) (s : State)
  : CallResultV2 :=
  if s.ended then .failure .
     AuctionAlreadyEnded
  else if ctx.timestamp < s.endTime then
    .failure .AuctionNotYetEnded
  else
    let newState := { s with
      ended := true
      pendingReturns := creditSeller s
      nftOwner := s.highestBidder
      -- BUG: highestBid intentionally
     unchanged
    }
    .success { newState := newState,
               events := [Event.AuctionEnded
                 s.highestBidder s.highestBid
    ] }

-- Newly induced by F1; this failing target
     joins the
-- existing non-blocking finalization
     obligations.
theorem endAuction_should_preserve_solvency
  (ctx : Context) (s : State) (out :
     TransitionOutputV2) :
  endAuction_v2 ctx s = .success out ->
  solvent out.newState := by
  sorry
```

Lean V2 excerpt: solvency added as a new obligation

Figure 16: **Round 1 (Lean).** The model adds solvency as a new proof target while retaining the earlier non-blocking-finalization obligation.

closing path must also preserve accounting consistency.

AttackAnalyst feedback — F_1

The next round must align the implementation with the intended post-state accounting. Once the seller credit is materialized, the transient winning-bid field must be cleared. The new formal target is therefore stronger than merely using pull payments: successful closure must preserve solvency.

C.4 Round 2 : Restored Closure and Solvency

Round 2: Candidate C_2

The final contract keeps the pull-payment structure, clears the stale winning-bid field, and restores both non-blocking closure and liability-consistent accounting under the chosen witness model.

At this point, as shown in Figure 17 and Figure 18, the closing path no longer contains a seller-dependent external call, and the stale winning-bid field is cleared when the seller credit is materialized. The Lean model can therefore check together

```
function endAuction() external {
    if (ended) revert AuctionAlreadyEnded();
    if (block.timestamp < endTime)
        revert AuctionNotYetEnded();

    ended = true;

    uint256 payout = highestBid;
    if (payout > 0) {
        pendingReturns[seller] += payout;
        highestBid = 0;    // critical fix
    }

    nft.transferFrom(address(this), highestBidder,
      tokenId);
    emit AuctionEnded(highestBidder, payout);
}
```

Solidity C_2: pull pattern plus accounting fix

Figure 17: **Round 2 (Solidity).** The final candidate keeps the pull-payment pattern and adds the accounting fix by clearing `highestBid`.

the obligations accumulated across previous rounds (Figure 18): non-blocking closure with respect to seller behavior, solvency of the post-state, and correct delivery of the NFT to the highest bidder.

Attacker witness — τ_2

```
=== Auction V3 Final Security Trace ===
seller pending after end: 10000000000000000000
highestBid after end: 0
contract balance: 10000000000000000000
recorded liability: 10000000000000000000
nft owner after end: 0x...0B0b
```

The same malicious seller used in Round 0 no longer blocks closure. The auction ends successfully, the bidder receives the NFT, and the contract's recorded liability exactly matches the 10 ETH escrowed in the contract. If the seller later calls `withdraw()` and rejects ETH again, that failure affects only the seller's own withdrawal attempt; it does not rewind the completed auction. The attack family discovered in the first round is therefore neutralized, and the accounting inconsistency discovered in the second round is repaired as well.

AttackAnalyst feedback — Stop criterion under the chosen model

At this point the main discovered attack family is blocked under the chosen witness model. The final state combines the liveness improvement of pull payments with the accounting repair needed for solvency. The appendix should still state the modeling boundary clearly: the proof is over a reduced closing witness, not a proof of every real-world auction interaction

```
def endAuction_v3 (ctx : Context) (s : State)
  : CallResultV3 :=
  if s.ended then .failure .
     AuctionAlreadyEnded
  else if ctx.timestamp < s.endTime then
    .failure .AuctionNotYetEnded
  else
    let payout := s.highestBid
    let creditedPending := fun a =>
      if a = s.seller then s.pendingReturns a
      + payout
      else s.pendingReturns a
    let newState := { s with
      ended := true
      pendingReturns := creditedPending
      highestBid := 0
      nftOwner := s.highestBidder }
    .success { newState := newState,
               events := [Event.AuctionEnded
                 s.highestBidder payout] }

-- Obligations discovered across previous
     rounds are now
-- checked together for the final transition.
theorem
     v3_is_non_blocking_wrt_seller_acceptance
      : ...
theorem
     v3_preserves_solvency_for_closing_witness
      : ...
theorem v3_transfers_nft_to_winner : ...
```

Lean V3 excerpt: carried-forward obligations are discharged

Figure 18: **Round 2 (Lean).** The final model checks the carried-forward obligations for non-blocking closure, solvency, and NFT delivery together.

pattern outside the modeled state transition.

C.5 Compact Summary Across Rounds

A compact summary of the round-by-round evolution of the LeVer loop is given in Table 3, including the coder output, verifier delta, attacker witness, and analyst feedback at each step.

C.6 Takeaway

Presented in the same order as the framework itself, the three rounds make the LeVer loop concrete in a simpler state space. The Coder does not merely patch syntax: each iteration changes what the closing path is allowed to mean. The Verifier does not merely check finished code: in every round it maintains and rechecks the current property set while progressively strengthening the formal object, moving from basic closure guards, to non-blocking finalization, to solvency and asset-delivery guarantees. The Attacker is not redundant with the Verifier because each trace reveals precisely which semantic assumption is still too weak.

The progression is also structurally parallel to the larger zap-in example. In Round 0, the attack reveals an interaction-level liveness bug: closure should not depend on seller-side ETH acceptance. In Round 1, once that issue is repaired, the next discovered bug is an accounting inconsistency: pull payments alone do not preserve solvency if stale debt remains in the state. In Round 2, the final contract satisfies both goals under the chosen witness model: auction closure is non-blocking, and the recorded liability matches the escrowed ETH. The AttackAnalyst is therefore the crucial bridge between concrete execution traces and the next formal property that the system must be able to state and eventually prove.

Round	Coder output	Verifier delta	Attacker witness	Analyst feedback
V1 / C_0	Pushes ETH to the seller during `endAuction`.	Baseline closure checks are present; the trace induces a new non-blocking-finalization target.	A reverting seller makes `endAuction` fail; `ended=0`, and both the NFT and 10 ETH remain locked.	Use pull payments so finalization no longer depends on seller behavior.
V2 / C_1	Credits `pendingReturns` and transfers the NFT, but leaves `highestBid` stale.	The non-blocking target is retained; solvency is added as a new accounting obligation.	Recorded liability is 20 ETH while contract balance is 10 ETH.	Clear the transient bid field once the seller credit is materialized.
V3 / C_2	Uses pull payment and sets `highestBid=0`.	Carried-forward obligations for non-blocking closure, solvency, and NFT delivery are checked together.	Closure succeeds; seller pending is 10 ETH, `highestBid=0`, and liability equals balance.	Main discovered attack families are blocked under the chosen witness model.

Table 3: **One row per iteration of the LeVer loop in the auction case study.**

D Complete Lean Artifact for the Zap-in Case Study

This subsection explains the final Lean artifact used in the zap-in case study. The artifact is the formal counterpart of the final Solidity candidate in the case-study appendix. It models the security-relevant ERC-20 zap-in path as a pure transition function, reifies token-transfer side effects as explicit data, and proves both the carried-forward asset-flow obligations and the final fair-value slippage guarantee.

The purpose of this artifact is not to give a full EVM semantics or a complete Curve implementation. Instead, it follows the semantic translation style used throughout LeVer: Solidity storage is represented as immutable Lean structures, Solidity `require`-style checks are represented as conditional guards, state mutation is represented as construction of a new state, and reverted executions are represented by `CallResult.failure`. A Lean theorem is a type, and its proof is a term inhabiting that type; therefore, once the proof scripts below type-check, the corresponding properties are machine-checked.

Imports, primitive abstractions, and observable outcomes. The artifact first imports Lean libraries and defines lightweight abstractions for addresses and token amounts. Both are modeled as natural numbers. This abstraction avoids irrelevant byte-level EVM details while preserving equality on addresses and arithmetic over token amounts, which are the ingredients needed for allowance, output, and slippage reasoning.

```
import Lean
import Std

-- 1. Basic types and error definitions

abbrev Address := Nat
abbrev Amount := Nat

def BPS_DENOMINATOR : Nat := 10000

inductive Error where
  | InsufficientAllowance
  | InsufficientInput
  | ContractStopped
  | SlippageToleranceExceeded
  | InvalidTokenAddress
  | Unauthorized
  deriving Repr, DecidableEq

inductive Event where
  | Approved (owner : Address) (spender :
     Address) (value : Amount)
  | ZapInExecuted (sender : Address) (
     inAmount : Amount) (outAmount : Amount)
  | ContractStatusChanged (isStopped : Bool)
  deriving Repr, DecidableEq
```

The constant `BPS_DENOMINATOR` fixes the unit for basis points. In this model, 10000 represents 100%, and a tolerance of 100 represents 1%. The `Error` type collects the possible failure reasons of the transition. The `Event` type records abstract event emissions. These are not intended to model the full EVM log system; they are included so that the transition output has the same observable components as the Solidity candidate.

External market model and execution context. The next part models the external market and the transaction context. This is where the final version differs from earlier static-price abstractions. The market state contains not only reserves, but also explicit token identities.

```

-- 2. External-world model

-- Repair: add explicit token identities to
     the market model.
```

```

structure MarketState where
  tokenIn    : Address -- Repair 1: the token
      accepted by the pool
  tokenOut   : Address
  reserveIn  : Amount
  reserveOut : Amount
  deriving Repr, DecidableEq

structure Context where
  sender : Address
  market : MarketState
  -- Repair 2: oracle-like fair-value
     information for dimension-safe pricing
  fairPrice      : Amount
  pricePrecision : Amount
  deriving Repr, DecidableEq

def get_amm_out (market : MarketState) (dx :
    Amount) : Amount :=
  let x := market.reserveIn
  let y := market.reserveOut
  if x + dx == 0 then 0 else
  (y * dx) / (x + dx)
```

The field `tokenIn` is essential for ruling out the fake-token attack discovered in the previous iteration. Without this field, the transition could reason only about an input amount, not about whether the paid asset was the asset accepted by the pool. The fields `fairPrice` and `pricePrecision` encode the fair-value baseline used for dimension-consistent slippage calculation. This is needed because the input and output assets may have different units and decimal conventions.

The function `get_amm_out` abstracts the pool response as a reserve-dependent output:

$$\texttt{out} = \frac{\texttt{reserveOut} \cdot \texttt{dx}}{\texttt{reserveIn} + \texttt{dx}}.$$

This is not a full Curve invariant. Its role in the case study is to remove the fixed-price stub and make the output depend on an explicit market state.

Contract state and reified side effects. The persistent contract state is represented by the `State` structure. The allowance table is modeled as a total function from owner and spender addresses to amounts.

```

-- 3. Contract state and transition
    structures

structure State where
  stopped : Bool
  goodwill : Amount
  allowances : Address -> Address -> Amount
  poolToken : Address
  owner : Address

structure TransferAction where
  fromAddr : Address
  toAddr   : Address
  token    : Address
  amount   : Amount
  deriving Repr, DecidableEq

structure TransitionOutput where
  newState : State
  events   : List Event
  transfers : List TransferAction

inductive CallResult where
  | success (out : TransitionOutput)
  | failure (err : Error)
```

This block is important for understanding how Lean represents Solidity side effects. A Solidity contract mutates storage and calls token contracts. The Lean model does not execute such calls directly. Instead, it reifies intended asset movements as values of type `TransferAction`. Each transfer action records the source, destination, token identity, and amount.

This design is what makes the earlier fair-exchange bug expressible. A model that only records "the user receives some amount" cannot prove that the user also paid the input token. In the final artifact, the transfer list can be inspected by theorems, so properties such as input payment, allowance consumption, and output delivery become formal obligations.

Administrative and approval transitions. The model next defines two auxiliary transitions. The first one changes the stopped flag and is restricted to the owner. The second one models token approval by updating the allowance function.

```

-- 4. Contract logic

def setStopped (newStatus : Bool) (ctx :
    Context) (s : State) : CallResult :=
  if ctx.sender != s.owner then
    CallResult.failure Error.Unauthorized
  else
    let newState := { s with stopped :=
     newStatus }
    CallResult.success {
      newState := newState,
      events := [Event.ContractStatusChanged
     newStatus],
      transfers := []
    }

def approveToken (spender : Address) (amount
    : Amount)
                 (ctx : Context) (s : State)
    : CallResult :=
  if spender = 0 then
    CallResult.failure Error.
     InvalidTokenAddress
  else
    let newAllowances := fun owner spdr =>
      if owner = ctx.sender /\ spdr = spender
      then amount
      else s.allowances owner spdr
    CallResult.success {
      newState := { s with allowances :=
     newAllowances },
      events := [Event.Approved ctx.sender
     spender amount],
      transfers := []
    }
```

The expression `{ s with stopped := newStatus }` constructs a new record by copying all fields of `s` except `stopped`. This is the

functional counterpart of a Solidity storage assignment.

The allowance update follows the same idea for mappings. A Solidity nested mapping such as `allowances[owner][spender]` is represented in Lean as a function `Address -> Address -> Amount`. Updating one entry means constructing a new function: where $\chi(o, s)$ abbreviates $o = \texttt{ctx.sender} \wedge s = \texttt{spender}$, and

$$\lambda o, s.\ \begin{cases} \texttt{amount}, & \chi(o, s), \\ \texttt{s.allowances}\ o\ s, & \text{otherwise.} \end{cases}$$

This functional representation is what later allows the allowance theorem to reduce by simplification.

Fair-value expected output. The final model separates the AMM spot output from the fair-value expected output. The expected output is computed using an oracle-like price and a precision parameter.

```
-- Repair 2: compute the fair-value expected
     output.
-- This models the front-end or oracle-side
     calculation used to derive minLiquidity.
def calc_expected_out
    (amountIn : Amount) (price : Amount) (
    prec : Amount) : Amount :=
  if prec == 0 then 0 else (amountIn * price)
     / prec
```

This definition is the formal fix for the dimension mismatch. Earlier versions implicitly compared output amounts against `amountIn`, which is only meaningful when both quantities use the same unit. In heterogeneous pairs such as WBTC/WETH, this is unsound. The final model first converts the input amount into an output-token-denominated expected amount, and then uses that value as the slippage baseline.

Main ERC-20 zap-in transition. The core transition is `ZapInWithERC20_Dynamic`. It follows a guard-action pattern: every invalid condition returns a failure, while the final success branch updates allowance and emits both sides of the asset flow.

```
-- Repaired ZapIn transition.
def ZapInWithERC20_Dynamic
    (inToken : Address) (inAmount : Amount) (
    minLiquidity : Amount)
    (ctx : Context) (s : State) : CallResult
    :=
  if s.stopped then
    CallResult.failure Error.ContractStopped
  else if inAmount = 0 then
    CallResult.failure Error.
    InsufficientInput
  else if s.allowances ctx.sender inToken <
    inAmount then
    CallResult.failure Error.
    InsufficientAllowance
  -- Repair 1: validate that the input token
    matches the market input token.
  else if inToken != ctx.market.tokenIn then
    CallResult.failure Error.
    InvalidTokenAddress
  else
    let actualOut := get_amm_out ctx.market
    inAmount
    if actualOut < minLiquidity then
      CallResult.failure Error.
     SlippageToleranceExceeded
    else
      let newAllowances := fun owner spdr =>
        if owner = ctx.sender /\ spdr =
     inToken then
          s.allowances owner spdr - inAmount
        else s.allowances owner spdr

      let newState := { s with allowances :=
     newAllowances }

      let actionPull : TransferAction :=
        { fromAddr := ctx.sender, toAddr :=
     999,
          token := inToken, amount :=
     inAmount }

      let actionPush : TransferAction :=
        { fromAddr := 999, toAddr := ctx.
     sender,
          token := s.poolToken, amount :=
     actualOut }

      CallResult.success {
        newState  := newState,
        events    := [Event.ZapInExecuted ctx.
     sender inAmount actualOut],
        transfers := [actionPull, actionPush]
      }
```

The first three guards check the paused state, zero input, and allowance. The fourth guard is the token-identity repair:

$$\texttt{inToken = ctx.market.tokenIn.}$$

This is what blocks the fake-token trace from the previous round. The fifth guard is the slippage check: if the AMM output is below `minLiquidity`, the transition fails.

The success branch has three effects. First, it constructs a new allowance function where the caller's allowance for the input token is reduced by `inAmount`. Second, it records an input-payment action from the user to the contract. Third, it records an output action from the contract to the user. The address `999` is an abstract placeholder for the modeled contract address. It is sufficient for this proof because the asset-flow properties only need to distinguish user-side and contract-side actions.

Loss measurement and safe minimum liquidity. The verification section defines the loss metric and the construction of a safe minimum-liquidity parameter.

```

-- 5. Formal verification: completeness and
     robustness theorems

def calc_loss_bps (expected : Amount) (actual
     : Amount) : Amount :=
  if actual >= expected then 0
  else (expected - actual) * BPS_DENOMINATOR
     / expected

def construct_safe_min_liquidity
    (expectedOut : Amount) (toleranceBps :
    Amount) : Amount :=
  let numerator := expectedOut * (
     BPS_DENOMINATOR - toleranceBps)
  if numerator == 0 then 0 else
  (numerator + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR
```

The function `calc_loss_bps` returns zero when the actual output is at least the expected output. Otherwise, it computes the relative loss in basis points:

$$\frac{(\texttt{expected} - \texttt{actual}) \cdot 10000}{\texttt{expected}}.$$

`construct_safe_min_liquidity` computes

$$\left\lceil \frac{\texttt{expectedOut} \cdot (10000 - \texttt{toleranceBps})}{10000} \right\rceil.$$

The rounding direction is conservative. Rounding down would weaken the user's protection, while rounding up preserves the desired lower bound.

Ceiling-division lemma. The main slippage proof needs a small arithmetic fact: multiplying a rounded-up quotient by the denominator gives at least the original numerator.

```
-- Arithmetic helper: ceil(n /d) * d > = n.
theorem ceil_div_mul_ge (n d : Nat) (h_d_pos
     : d > 0) :
  ((n + d - 1) / d) * d >= n :=
by
  match n with
  | 0 => simp
  | n' + 1 =>
    have h_lhs :
        ((n' + 1 + d - 1) / d) * d =
        (n' / d + 1) * d := by
      rw [Nat.add_comm n' 1, Nat.add_assoc,
          Nat.add_comm 1, Nat.add_sub_cancel]
      rw [Nat.add_div_right _ h_d_pos]
    rw [h_lhs]
    rw [Nat.add_mul, Nat.one_mul]
    conv => rhs; rw [<- Nat.div_add_mod n' d]
    rw [Nat.mul_comm d (n' / d)]
    rw [Nat.add_assoc]
    apply Nat.add_le_add_left
    exact Nat.succ_le_of_lt (Nat.mod_lt _
     h_d_pos)
```

The proof splits on `n`. The zero case is immediate. For the successor case, the proof rewrites the ceiling expression into a form involving `n' / d + 1`. It then uses the standard division decomposition:

$$n' = (n'/d) \cdot d + (n' \bmod d).$$

Since `Nat.mod_lt` gives $n' \bmod d < d$, one additional copy of d is enough to cover the remainder. This proves the inequality needed later when `minLiquidity` is multiplied back by 10000.

Main slippage theorem: statement. The central theorem states that if `minLiquidity` is constructed from the fair-value expected output, then any successful execution has loss at most the user-specified tolerance.

```
-- Main theorem: robustness of the slippage
     mechanism.
-- The minimum-liquidity bound is computed
     from expectedOut, and any successful
-- execution has bounded loss relative to
     that fair-value benchmark.
theorem slippage_mechanism_is_robust
  (inToken : Address)
  (amountIn : Amount)
  (toleranceBps : Amount)
  (market : MarketState)
  (s : State)
  -- Oracle-like fair-value parameters.
  (fairPrice : Amount)
  (pricePrecision : Amount)
  (h_pos : amountIn > 0)
  (h_tol : toleranceBps <= BPS_DENOMINATOR)
  -- Assumptions: nonzero oracle precision
     and valid input token.
  (h_prec_pos : pricePrecision > 0)
  (h_token_valid : inToken = market.tokenIn)
     :

  -- Front-end calculation: compute
     expectedOut before constructing
     minLiquidity.
  let expectedOut := calc_expected_out
     amountIn fairPrice pricePrecision
  let minLiq := construct_safe_min_liquidity
     expectedOut toleranceBps

  let ctx : Context :=
    { sender := 1, market := market,
      fairPrice := fairPrice, pricePrecision
     := pricePrecision }
  let result := ZapInWithERC20_Dynamic
     inToken amountIn minLiq ctx s

  match result with
  | CallResult.failure _ => True
  | CallResult.success _ =>
      let actualOut := get_amm_out market
     amountIn
      -- Proof goal: loss relative to
     expectedOut is bounded by tolerance.
      calc_loss_bps expectedOut actualOut <=
     toleranceBps :=
```

The theorem has four explicit semantic assumptions. The input amount is positive, the tolerance is at most 100%, the oracle precision is nonzero, and the input token matches the market's accepted token. The result is phrased by case analysis on the transition result. If the call fails, the theorem requires only `True`, because a reverted transaction commits no unsafe state. If the call succeeds, the theorem proves the basis-point loss bound.

Main slippage theorem: unfolding and guard splitting. The proof begins by unfolding the transition and arithmetic definitions. After unfolding,

the proof goal has the same branch structure as the transition function.

```
by
  dsimp
  -- Unfold all relevant definitions.
  dsimp [ZapInWithERC20_Dynamic,
     calc_loss_bps,
         construct_safe_min_liquidity,
     calc_expected_out]

  -- Guard 1: stopped contract.
  by_cases hs : s.stopped = true
  ů simp [hs]

  -- Guard 2: zero input amount.
  by_cases hA0 : amountIn = 0
  ů simp [hs, hA0]

  -- Guard 3: insufficient allowance.
  by_cases hAllow : s.allowances 1 inToken <
     amountIn
  ů simp [hs, hA0, hAllow]

  -- Guard 4: token validation.
  -- The invalid-token branch contradicts
     h_token_valid.
  by_cases hToken : inToken != market.tokenIn
  ů have : inToken = market.tokenIn :=
     h_token_valid
    contradiction
```

Each guard split corresponds to a Solidity-style precondition. If the contract is stopped, the input is zero, or allowance is insufficient, the transition fails, and the theorem is discharged by `simp`. The token-invalid branch is impossible because the theorem assumes `h_token_valid`. This is where the final token-identity repair is used inside the proof.

Main slippage theorem: slippage branch analysis. After the basic guards are eliminated, the proof introduces abbreviations for the AMM output, expected output, and numerator used by the minimum-liquidity formula. It then splits on whether the slippage guard fires.

```
  -- Prepare abbreviations used in the
     arithmetic proof.
  let actualOut := get_amm_out market
     amountIn
  let expectedOut := (amountIn * fairPrice) /
      pricePrecision
  let N := expectedOut * (BPS_DENOMINATOR -
     toleranceBps)

  -- Guard 5: slippage check, actualOut <
     minLiquidity.
  -- This is the same proof structure as the
     previous version, except that
  -- amountIn has been replaced by
     expectedOut.
  by_cases hSlip :
      actualOut <
        (if (N == 0) = true then 0
         else (N + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR)
  ů
    -- Positive branch: slippage protection
     triggers, so the call fails.

    -- Rewrite the nested if expression into
     the shape expected by simp.
    have hSlip_pos' :
        get_amm_out market amountIn <
          (if expectedOut * (BPS_DENOMINATOR
     - toleranceBps) = 0 then 0
           else (expectedOut * (
     BPS_DENOMINATOR - toleranceBps)
                  + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) := by
      simpa [actualOut, N, expectedOut] using
      hSlip

    have hprec_ne0 : pricePrecision != 0 :=
      Nat.ne_of_gt h_prec_pos

    have hSlip_pos'' :
        get_amm_out market amountIn <
          (if
              (amountIn * fairPrice /
     pricePrecision) *
                 (BPS_DENOMINATOR -
     toleranceBps) = 0 then
            0
           else
            ((amountIn * fairPrice /
     pricePrecision) *
                   (BPS_DENOMINATOR -
     toleranceBps) +
                 BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) := by
      simpa [expectedOut, hprec_ne0] using
     hSlip_pos'

    simp [hs, hA0, hAllow, hToken, hprec_ne0,
      hSlip_pos'']
```

The positive branch corresponds to a transaction that fails the slippage check. Since failed transitions are safe in the theorem statement, the proof only needs to rewrite the condition into the syntactic form expected by the unfolded goal. The assumption `h_prec_pos` is converted into `pricePrecision != 0`, allowing the expected-output definition to simplify.

Main slippage theorem: extracting the success-side lower bound. The negative slippage branch is the substantive case. Here, the slippage guard does not fire, so the transition reaches the success branch.

```

    -- Negative branch: slippage protection
     does not trigger.
    -- Therefore, the transition reaches the
     success branch.

    have hSlip_neg' :
        ň get_amm_out market amountIn <
          (if expectedOut * (BPS_DENOMINATOR
     - toleranceBps) = 0 then 0
           else (expectedOut * (
     BPS_DENOMINATOR - toleranceBps)
                  + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) := by
      simpa [actualOut, N, expectedOut] using
      hSlip

    -- This simplification enters the success
      branch of the match.
    simp [hs, hA0, hAllow, hToken]

    -- 1) Extract the core inequality:
     actualOut >= minLiquidity.

    have h_ge_min0 :
        actualOut >=
          (if (N == 0) = true then 0
           else (N + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) :=
      Nat.le_of_not_lt hSlip
```

The hypothesis `h_ge_min0` is the formal version of the runtime slippage check:

$$\texttt{actualOut} \geq \texttt{minLiquidity}.$$

This is the bridge from the executable guard to the later arithmetic proof.

Main slippage theorem: applying the ceiling lemma. The proof now converts the lower bound on `actualOut` into the stronger inequality needed for relative-loss reasoning:

$$\texttt{actualOut} \cdot 10000 \geq \texttt{expectedOut} \cdot (10000 - \texttt{toleranceBps}).$$

```
    -- 2) Prove actualOut * denominator >= N
     using the ceiling lemma.

    have h_O_ge_N : actualOut *
     BPS_DENOMINATOR >= N := by
      by_cases hN0 : N = 0
      · rw [hN0]
        exact Nat.zero_le _
      · by_cases hb : (N == 0) = true
        · have hb_iff : (N == 0) = true <-> N
       = 0 := by
            simpa using (Nat.beq_eq_true N 0)
          exact False.elim (hN0 (hb_iff.mp hb
    ))
        · have h_ge_ceil :
              actualOut >=
                (N + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR := by
            simpa [hb] using h_ge_min0
          have h_ceil :=
            ceil_div_mul_ge N BPS_DENOMINATOR
      (by decide)
          have h_mul :=
            Nat.mul_le_mul_right
     BPS_DENOMINATOR h_ge_ceil
          exact Nat.le_trans h_ceil h_mul
```

If $N = 0$, the inequality is immediate. Otherwise, the proof first shows that `actualOut` is at least the ceiling-divided value. The helper theorem `ceil_div_mul_ge` then proves that multiplying that lower bound by 10000 is sufficient to cover N. This step is the mathematical core of the slippage proof.

Main slippage theorem: simplifying the success goal. The remaining proof is mostly proof engineering. The unfolded goal contains nested conditionals from `calc_expected_out` and the slippage guard. The proof removes the zero-precision branch and names the remaining slippage condition so that `simp` can reduce the goal to the final loss inequality.

```
    -- 3) Split into no-loss and loss cases
     relative to expectedOut.

    -- Use pricePrecision != 0 to eliminate
     the precision-zero branch.
    have hprec_ne0 : pricePrecision != 0 :=
      Nat.ne_of_gt h_prec_pos

    -- Rewrite the non-triggered slippage
     hypothesis into the form that
    -- contains the same if(pricePrecision =
     0) expression as the goal.
    have hSlip_goal :
        ň get_amm_out market amountIn <
          (if
              (if pricePrecision = 0 then 0
               else amountIn * fairPrice /
     pricePrecision) *
                (BPS_DENOMINATOR -
     toleranceBps) = 0 then
            0
           else
            ((if pricePrecision = 0 then 0
              else amountIn * fairPrice /
     pricePrecision) *
                (BPS_DENOMINATOR -
     toleranceBps) +
              BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) := by
      -- Use hprec_ne0 to reduce if(
     pricePrecision = 0) to the else branch,
      -- and then fold it back through
     expectedOut.
      simpa [expectedOut, hprec_ne0] using
     hSlip_neg'

    -- Now the outer if/match in the goal has
      the same shape and can be simplified.
    simp [hprec_ne0]

    -- At this point, the goal still contains
      the final slippage if/match.
    -- We introduce a proposition cond so
     simp can use its negation directly.
    have hSlip_final :
        ň get_amm_out market amountIn <
          (if amountIn * fairPrice /
     pricePrecision *
                (BPS_DENOMINATOR -
     toleranceBps) = 0 then
              0
           else
              (amountIn * fairPrice /
     pricePrecision *
                  (BPS_DENOMINATOR -
     toleranceBps)
                + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR) := by
      -- hSlip_neg' uses expectedOut * (D - T
     ); unfold expectedOut.
      simpa [expectedOut] using hSlip_neg'

    -- Name the condition explicitly so simp
     can reduce the success branch.
    let cond : Prop :=
      get_amm_out market amountIn <
        (if amountIn * fairPrice /
     pricePrecision *
              (BPS_DENOMINATOR - toleranceBps
     ) = 0 then
            0
         else
            (amountIn * fairPrice /
     pricePrecision *
                (BPS_DENOMINATOR -
     toleranceBps)
              + BPS_DENOMINATOR - 1) /
     BPS_DENOMINATOR)

    have hcond : ň cond := by
      simpa [cond] using hSlip_final

    -- This removes the remaining match/if
     and exposes the loss-bound goal.
    simp [cond, hcond]
```

At this point, the proof goal has been reduced to the basis-point loss expression. The complicated branch structure of the transition has been eliminated, and only arithmetic remains.

Main slippage theorem: final loss-bound arithmetic. The proof introduces shorter names E and O for the expected output and actual output. It then splits on whether the trade has no loss relative to the fair-value benchmark.

```
    -- The remaining goal is:
    --   (if expectedOut <= actualOut then 0
    --    else (expectedOut - actualOut) *
     BPS_DENOMINATOR / expectedOut)
    --   <= toleranceBps.
    let E : Nat := amountIn * fairPrice /
     pricePrecision
    let O : Nat := get_amm_out market
     amountIn

    -- Align the local abbreviations with the
      previous let-bound names.
    have hE : expectedOut = E := by rfl
    have hO : actualOut = O := by rfl

    by_cases h_profit : O >= E
    · -- No-loss branch.
      have h_le_goal : E <= O := h_profit
      -- The if-expression reduces to the
     zero-loss branch.
      simp [E, O, h_le_goal]

    · -- Loss branch.
      have h_not_le_goal : ¬ E <= O := by
        intro hle
        exact h_profit hle
      -- Reduce the if-expression to the
     nonzero loss expression.
      simp [E, O, h_not_le_goal]

      -- The remaining goal is:
      --   (E - O) * BPS_DENOMINATOR / E <=
     toleranceBps.

      apply Nat.div_le_of_le_mul
      rw [Nat.mul_sub_right_distrib]
      apply Nat.sub_le_of_le_add

      -- Reuse the previous inequality chain
     after substituting E and O.
      apply Nat.le_trans (m := N + E *
    toleranceBps)
      · -- Prove BPS_DENOMINATOR * E <= N + E
      * toleranceBps.
        -- Here N = expectedOut * (D - T),
    and expectedOut = E.
        dsimp [N]
        simp [hE, E]
        rw [Nat.mul_sub_left_distrib]
        apply Nat.le_add_of_sub_le
        exact Nat.le_refl _
      · -- Prove N + E * toleranceBps <=
    BPS_DENOMINATOR * O + E * toleranceBps.
        -- h_O_ge_N states actualOut * D >= N
    , and actualOut = O.
        have h1 :
            N + E * toleranceBps <=
              O * BPS_DENOMINATOR + E *
    toleranceBps :=
          Nat.add_le_add_right
            (by simpa [hO, O] using h_O_ge_N)
            (E * toleranceBps)
        simpa [Nat.add_comm, Nat.
    add_left_comm,
              Nat.add_assoc, O] using h1
```

If $O \geq E$, the loss is zero and the theorem is immediate. Otherwise, the target is

$$\frac{(E - O) \cdot 10000}{E} \leq \texttt{toleranceBps}.$$

Lean proves this by applying `Nat.div_le_of_le_mul`, which reduces a division inequality to a multiplication inequality. The key inequality from the previous step,

$$O \cdot 10000 \geq E \cdot (10000 - \texttt{toleranceBps}),$$

is then rearranged into the desired loss bound:

$$(E - O) \cdot 10000 \leq E \cdot \texttt{toleranceBps}.$$

This completes the main robustness proof.

Carried-forward theorem: payment is enforced. The first carried-forward execution theorem proves that every successful zap-in contains an input-payment action from the caller. This is the formal version of the property introduced after the original faucet attack.

```
-- Fair-exchange theorem: a successful zap-in
     must include an input-payment action.
theorem zap_in_enforces_payment
  (inToken : Address) (amountIn : Amount) (
    minLiquidity : Amount)
  (ctx : Context) (s : State) (out :
    TransitionOutput) :
  ZapInWithERC20_Dynamic inToken amountIn
    minLiquidity ctx s =
    CallResult.success out ->
  exists act, act  out.transfers /\
        act.fromAddr = ctx.sender /\
        act.token = inToken /\
        act.amount = amountIn :=
by
  intro h_success
  dsimp [ZapInWithERC20_Dynamic] at h_success
  -- Split all guards, including the added
    token-validation guard.
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success
  · contradiction -- Slippage-failure branch.
  · injection h_success with h_eq
    rw [<- h_eq]
    simp
```

The proof pattern is direct. It unfolds the transition and splits over all guards. Every failure branch contradicts the assumption that the result is `CallResult.success out`. In the success branch, `injection` extracts equality of the successful outputs, and `simp` finds `actionPull` in the two-element transfer list.

Carried-forward theorem: allowance is consumed. The second execution theorem proves that the successful transition reduces the caller's allowance by exactly `amountIn`.

```
-- Allowance-consumption theorem.
theorem zap_in_reduces_allowance
  (inToken : Address) (amountIn : Amount) (
    minLiquidity : Amount)
  (ctx : Context) (s : State) (out :
    TransitionOutput) :
  ZapInWithERC20_Dynamic inToken amountIn
    minLiquidity ctx s =
    CallResult.success out ->
  s.allowances ctx.sender inToken >= amountIn
    ->
```

```
  out.newState.allowances ctx.sender inToken
    =
    s.allowances ctx.sender inToken -
     amountIn :=
by
  intro h_success h_allowance_enough
  dsimp [ZapInWithERC20_Dynamic] at h_success
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success
  ů contradiction
  ů injection h_success with h_eq
    rw [<- h_eq]
    simp
```

The proof again eliminates all failure branches. Once the success branch is isolated, the post-state allowance is exactly the functional update created in `ZapInWithERC20_Dynamic`. Evaluating that function at `ctx.sender` and `inToken` reduces to the subtraction in the theorem statement.

Carried-forward theorem: output action exists. The third execution theorem proves that a successful zap-in contains an output action sending the AMM-computed output amount to the caller.

```
-- Output-atomicity theorem: a successful zap
    -in must include the output action.
theorem zap_in_guarantees_output_action
  (inToken : Address) (amountIn : Amount) (
    minLiquidity : Amount)
  (ctx : Context) (s : State) (out :
    TransitionOutput) :
  ZapInWithERC20_Dynamic inToken amountIn
    minLiquidity ctx s =
    CallResult.success out ->
  let expectedOut := get_amm_out ctx.market
    amountIn
  exists act, act  out.transfers /\
        act.toAddr = ctx.sender /\
        act.token = s.poolToken /\
        act.amount = expectedOut :=
by
  intro h_success
  dsimp [ZapInWithERC20_Dynamic] at h_success
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success <;> try contradiction
  split at h_success
  ů contradiction
  ů injection h_success with h_eq
    rw [<- h_eq]
    simp
```

This theorem is the output-side counterpart of the payment theorem. It states that successful executions not only pull the user's input asset, but also contain the intended output transfer to the caller. The proof has the same structure: unfold the transition, discard impossible failure branches, and simplify the successful transfer list.

Summary of the Lean artifact. The final Lean artifact proves two classes of properties. The first class is execution integrity: successful calls include an input-payment action, consume allowance, and include the output action. These are the properties introduced after the early asset-flow failures. The second class is economic robustness: under explicit assumptions about token identity, positive input, nonzero price precision, and bounded tolerance, any successful call respects the user's slippage tolerance relative to a fair-value expected output.

The result is a model-level guarantee. It does not claim to verify all Curve internals, all ERC-20 implementations, or the full EVM. Rather, it certifies the specific transition system used in the case study and shows how attack-induced obligations are discharged as Lean proof scripts.