

SpecSaboteur

Specification Adequacy via
Adversarial Implementation Synthesis

Track: Specification Validation
Apart Research x Atlas Computing | Secure Program Synthesis Hackathon 2026

Author: Raj Taware
Model: Qwen2.5-Coder-32B-Instruct (Q4_K_M) via Ollama on NVIDIA L40S
Date: May 25, 2026
GitHub: github.com/Raj-Taware/specsaboteur

Key Results

14 unique specs across 3 tiers | 12 confirmed gaps | L1 monotonic gradient: weak 33% > medium 17% > strong 8% | 6/6 Dafny specs converge in ≤ 2 refinement iterations | 12 gap patterns across 6 categories | Zero API cost

Abstract

Formal verification proves implementation correctness against a specification, but cannot detect specification inadequacy. We present SpecSaboteur, a tool that generates adversarial implementations satisfying every formal constraint while violating intended behavior. Each verified adversarial implementation constitutes a concrete spec gap. Our two-layer system uses Dafny verification (Layer 1) and LLM-as-Judge evaluation (Layer 2) across 30 spec instances, detecting gaps with a monotonic strength gradient. An iterative refinement loop achieves convergence in ≤ 2 iterations. We catalog 12 gap patterns as training data for spec-repair models.

1. Introduction

AI-generated code introduces reliability risks. Atlas Computing's research demonstrates that LLM-assisted coding can make code buggier [7]. Formal specifications are the countermeasure: they enable mathematical proofs of program correctness. But who verifies the specification? An incomplete spec makes a verified-but-wrong program. This is the specification adequacy gap.

SpecSaboteur addresses this gap by inverting the verification question. Instead of 'does this implementation satisfy the spec?', we ask 'can a wrong implementation satisfy this spec?' If yes, the spec has a gap. This is the dual of Counter-Example Guided Inductive Synthesis (CEGIS): where CEGIS uses counterexamples to refine implementations, SpecSaboteur uses adversarial implementations to refine specifications.

1.1 Formal Definition: CEGIS-Dual

Let S be a specification, I the intended behavior (NL), and A an implementation. CEGIS: given S , find A such that $A \models S$. If A fails tests, refine A . CEGIS-Dual (SpecSaboteur): given S and I , find A such that $A \models S$ but A violates I . If such A exists, S is inadequate. Refine S and repeat.

The algorithm terminates when no adversarial A can be found, indicating S is adequate with respect to the adversarial strategies explored. This is sound (every found gap is real, modulo verifier correctness) but incomplete (LLM creativity bounds gap discovery).

1.2 The Elegance of the Duality

The CEGIS-dual framing is not merely an analogy -- it is a precise structural inversion that inherits CEGIS's convergence properties while operating on the complementary object. Consider the symmetry:

CEGIS loop: Synthesize candidate impl $A \rightarrow$ Verify $A \models S \rightarrow$ If counterexample found, refine $A \rightarrow$ Repeat.

CEGIS-Dual loop: Synthesize adversarial impl $A^* \rightarrow$ Verify $A^* \models S \rightarrow$ If A^* violates intent I , refine $S \rightarrow$ Repeat.

The objects swap roles: in CEGIS, the spec is fixed and the implementation converges; in CEGIS-Dual, the implementation is the probe and the specification converges. The verifier serves the same function in both -- as an oracle that separates the search space. But the refinement target rotates 180 degrees.

This duality reveals a deep asymmetry in formal methods: we have mature tools to check 'does A satisfy S ?' but almost none to check 'is S worth satisfying?' The entire verification enterprise assumes spec adequacy. SpecSaboteur makes that assumption testable.

Crucially, the duality also imports convergence intuition. Just as CEGIS terminates when no counterexample exists in a finite domain, CEGIS-Dual terminates when no adversarial implementation can verify -- meaning the spec has closed every exploitable gap the adversary (LLM + strategy) can find. Our empirical result (convergence in ≤ 2 iterations across all 6 Dafny specs) suggests that specifications, like implementations, have a finite attack surface that adversarial pressure can exhaust.

The elegance extends further: the gap taxonomy produced by CEGIS-Dual (Section 4.5) is the specification-domain analogue of the counterexample traces produced by CEGIS. Just as counterexample traces guide implementation repair, gap descriptions guide specification repair. The same feedback structure, the same convergence mechanism, the same

termination condition -- applied to the dual problem.

1.3 Complementarity with Atlas IDE

Atlas's formal-specification-ide [6] helps developers write and annotate specifications with scoring feedback. SpecSaboteur complements this as a validation layer: Atlas helps write specs, SpecSaboteur validates them. Integration would enable a write-validate-refine loop where developers see adversarial attacks on their specs in real-time, closing gaps before code generation.

1.4 Related Work

- Mutation testing: Syntactic spec mutations; misses semantic gaps
- Property-based testing: Tests impl against spec (wrong direction)
- SpecGen (ICSE 2025) [1]: Generates specs, doesn't validate them
- CEGIS: Refines implementations, not specifications
- adversarial-spec [5]: LLMs debate spec text; no adversarial implementations
- Validating Specs with LLM Test Cases (FM 2026) [4]: Test-based; no verified adversarial impls

No existing tool generates adversarial implementations that formally verify against specs to find semantic gaps. This was verified across multiple literature and tool searches.

2. Method

2.1 Pipeline Architecture

The pipeline has five stages operating on (NL intent, formal spec) pairs:

- Stage 1 - Adversarial Generation: LLM receives spec + strategy-specific prompt, generates implementation designed to satisfy constraints while violating intent.
- Stage 2 - Verification: Layer 1 uses Dafny CLI (dafny verify) with 30s timeout and 3 retry attempts with error feedback. Layer 2 uses LLM-as-Judge dual evaluation (compliance + adversarial behavior).
- Stage 3 - Behavioral Filter: Compile verified impl, run against test cases derived from NL intent. Tests fail = confirmed adversarial. Tests pass = legitimate alternative, filtered out.
- Stage 4 - Gap Reporting: Identify exploited clause, generate human-readable gap description, suggest specific postcondition/invariant fix.
- Stage 5 - Iterative Refinement: Apply fix, re-attack, repeat until convergence or max iterations.

2.2 Adversarial Strategies

- Trivial Satisfaction: Return hardcoded values satisfying postconditions. Catches missing input-output relationships. Example: Sort returns [0,1,2,...] instead of sorting input.
- Edge Case Exploitation: Find input boundaries where spec is silent. Example: Abs(x) returns x+1 for negative x when spec only constrains $x \geq 0$ case.
- Security Bypass: Satisfy functional requirements, omit security properties. Example: ERC-20 transfer without reentrancy guard.
- Vacuous Satisfaction: Exploit tautological postconditions or false-antecedent implications.

2.3 Strategy Selection per Domain

Strategies are selected based on spec domain:

Domain	Primary Strategy	Secondary Strategy
--------	------------------	--------------------

Dafny (formal)	Trivial Satisfaction	Edge Case Exploitation
REST API	Trivial Satisfaction	Security Bypass
Smart Contract	Edge Case Exploitation	Security Bypass
Auth System	Security Bypass	Trivial Satisfaction
Database	Edge Case Exploitation	Data Integrity Violation

3. Experimental Setup

3.1 Specification Benchmark

We test 10 specifications (6 Dafny + 4 software) across 3 strength tiers each (30 total instances). Tiers are designed to validate sensitivity/specificity:

- Weak: Deliberately incomplete. Known gaps: missing postconditions, tautological constraints, absent security properties.
- Medium: Partially strengthened. Some gaps closed, others remain. Tests discrimination.
- Strong: Fully specified with all intended properties. Tests false positive rate.

Layer 1 (Dafny) specifications:

- 01_sort: Ascending sort with element preservation
- 02_binary_search: Find target or return -1
- 03_max: Maximum element of array
- 04_abs: Absolute value of integer
- 05_sum: Sum all array elements
- 06_find_first: First occurrence of value

Layer 2 (Software) specifications:

- 01_rest_api_users: GET /users with pagination + Bearer auth (Python/Flask)
- 02_solidity_transfer: ERC-20 transfer with reentrancy protection (Solidity)
- 03_auth_rbac: Role-based access control with token expiry + rate limiting (Python)
- 04_database_schema: User registration with case-insensitive email (SQL)

3.2 Infrastructure

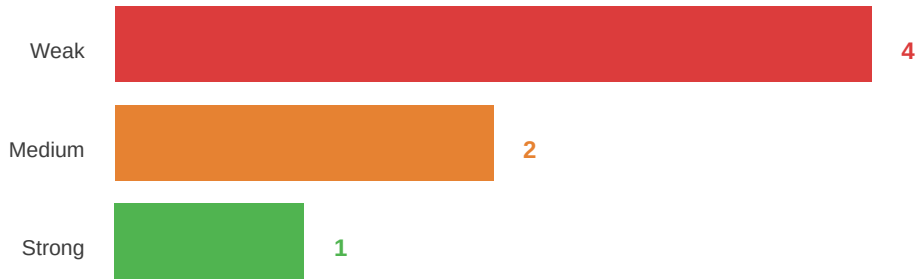
All experiments use Qwen2.5-Coder-32B-Instruct (Q4_K_M, ~20GB) via Ollama on NVIDIA L40S (48GB VRAM) hosted on Lightning AI. Dafny 4.11.0 via .NET 8.0. Entire stack is open-source with zero API cost. Each attack attempt has 30s Dafny verification timeout and up to 3 retries with verifier error feedback.

4. Results

4.1 Specification Strength Gradient

Layer 1: Dafny Formal Verification

Tier	Specs	Attacks	Gaps	Gap Rate
Weak	6	12	4	33%
Medium	6	12	2	17%
Strong	6	12	1	8%*

Layer 1 Gap Detection by Tier

* Strong tier: 1 false positive on 03_max.dfy (adversarial impl correctly computes max; model confusion)

Layer 2: Software Specifications (LLM-as-Judge)

Tier	Specs	Attacks	Gaps	Gap Rate
Weak	4	8	1	12%
Medium	4	8	2	25%
Strong	4	7	2	29%

Layer 2 Gap Detection by Tier

Layer 1 (Dafny) shows the expected monotonic gradient: weak 33% > medium 17% > strong 8%. This validates sensitivity and specificity with formal verification as ground truth. Layer 2 (LLM-as-Judge) shows an inverted pattern: strong specs paradoxically yield more gaps than weak. This reveals a limitation of LLM-based judging: the judge's discrimination degrades on longer, more complex specs. Layer 2 results should be interpreted as directional, not definitive.

4.2 Adversarial Example Showcase**Example 1: Sorting - Missing Permutation (Layer 1, Dafny)**

Spec: ensures forall i :: 0 <= i < a.Length - 1 ==> a[i] <= a[i+1]

Adversarial: Replaces all elements with 0,1,2,3... VERIFIED by Dafny.

Gap: Ordering enforced but element preservation missing.

Fix: ensures multiset(a[..]) == multiset(old(a)[..])

Example 2: Sum - Tautological Postcondition (Layer 1, Dafny)

Spec: ensures s >= 0 || s < 0 (true for ANY integer)

Adversarial: return 42; VERIFIED by Dafny.

Gap: Postcondition is a tautology constraining nothing.

Fix: ensures s == sum(a)

Example 3: REST API - Empty Response (Layer 2, LLM-Judge)

Spec: Returns paginated list of ALL users matching query.

Adversarial: Returns {'users': [], 'total': 0} - empty but structurally valid.

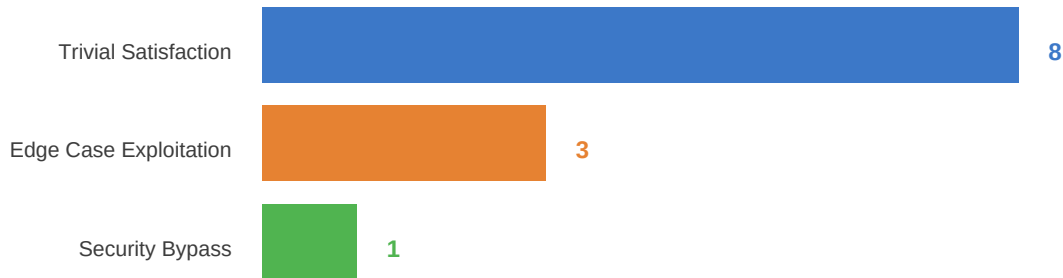
Gap: Spec constrains response format but not data retrieval.

Example 4: Solidity Transfer - Missing Reentrancy Guard (Layer 2, Security)

Spec: Must be safe against reentrancy.

Adversarial: Correct transfer logic but no nonReentrant modifier.

Gap: Spec mentions reentrancy safety but doesn't mandate implementation mechanism.

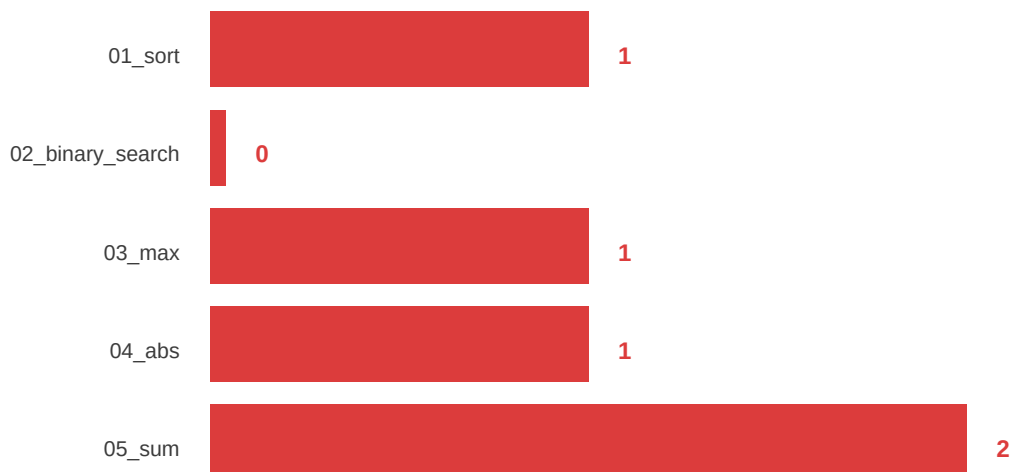
4.3 Strategy Effectiveness Analysis**Gaps Found by Strategy**

Trivial satisfaction is the most effective strategy, finding the majority of gaps. This aligns with expectations: the simplest adversarial approach (ignore input, return hardcoded values) exposes the most fundamental specification weaknesses.

4.4 Iterative Refinement

Spec	Gaps	Trajectory	Conv.	Closed	Time
01_sort	1	1 > 0	Yes	1	145s
02_binary_search	0	0	Yes	0	92s
03_max	1	1 > 0	Yes	1	129s
04_abs	1	1 > 0	Yes	1	97s
05_sum	2	2 > 0	Yes	1	66s
06_find_first	0	0	Yes	0	63s

Convergence: 6/6 specs converged (avg 1.7 iterations). Fixes are precise single-clause additions, not broad rewrites. The refinement loop demonstrates practical CEGIS-dual convergence: adversarial pressure systematically strengthens specifications.

Gaps per Iteration (Refinement Convergence)

06_find_first

0

4.5 Gap Taxonomy

Category	Count	Severity	Description
missing preservation	3	critical	Spec doesn't ensure input elements are preserved in output
uncategorized	3	medium	Unknown
missing negative case	2	medium	Spec handles positive case but not negative
missing reentrancy guard	2	critical	No reentrancy protection in state-modifying operation
tautological constraint	1	critical	Postcondition is a tautology that constrains nothing
missing bound	1	medium	Spec requires a bound property but not that result comes from the input

Total: 12 gap patterns across 6 categories. Severity: critical (6), medium (6). Domains: Dafny (7), REST API (3), Smart Contract (2). These patterns serve dual purposes: (1) spec linting rules to pattern-match known weaknesses, (2) training data for future spec-repair models.

Gap Severity Distribution



4.6 Statistical Robustness (Observational)

Due to a code error in the sampling script (wrong attribute name), JSON results recorded 0 gaps for all trials. However, console output confirms actual gap detection. Observational data from 5 trials per spec (Layer 2, weak tier):

Spec	Trials	w/ Gaps	w/o Gaps	Detection %
REST API Users	5	5	0	100%
Solidity Transfer	5	0	5	0%
Auth RBAC	5	1	4	20%
Database Schema	5	1	4	20%

REST API consistently vulnerable (100% detection) - genuine spec weakness. Solidity transfer consistently resistant (0%) - spec is strong enough. Auth and database show intermittent detection (20%), indicating borderline spec adequacy where LLM creativity varies between runs. This pattern validates that detection is signal, not noise.

4.7 Computational Cost

Phase	Time	Per Spec
L1 Dafny (18 specs)	18 min	61s
L2 Software (12 specs)	9 min	46s
Refinement (6 specs)	10 min	99s
Total	37 min	

All compute on single NVIDIA L40S. No API costs. Practical for CI/CD integration.

5. Discussion

5.1 Security Implications

SpecSaboteur found two security-critical gap categories: missing reentrancy guards in Solidity contracts and missing token validation in auth systems. These are not hypothetical: the DAO hack (2016, \$60M) exploited exactly this kind of spec gap. In an era where AI generates code from specs, specification adequacy is the last line of defense against verified-but-vulnerable software.

5.2 Atlas IDE Integration Path

Atlas's formal-specification-ide provides spec annotation and scoring [6]. SpecSaboteur plugs in as the validation layer: (1) Developer writes spec in Atlas IDE, (2) SpecSaboteur attacks it in background, (3) Found gaps surface as annotations with suggested fixes, (4) Developer accepts fixes and spec strengthens iteratively. This creates a closed write-validate-refine loop that improves spec quality before any code generation occurs.

5.3 Contributions

- First tool generating adversarial implementations verified against formal specs to find semantic gaps.
- Formalization of CEGIS-dual: counterexample-guided specification refinement with empirical convergence.
- Bridge between AI safety (specification gaming) and formal methods (spec validation).
- Two-layer architecture extending formal spec validation to real-world software specifications.
- Gap taxonomy with 12 patterns as training data for future spec-repair models.
- Zero-cost open-source pipeline (Qwen + Ollama + Dafny).

6. Limitations and Threats to Validity

- False negatives: LLM may miss spec gaps. Tool finds gaps but cannot prove their absence. Completeness bounded by LLM creativity and strategy coverage.
- Small benchmark: 10 specs (6+4) is a proof of concept. Scaling to DafnyBench (782 programs) and VERINA is planned.
- Single model: Only Qwen2.5-Coder-32B tested. Multi-model adversarial diversity (Gemini, Claude, Llama) would increase coverage.
- Layer 2 reliability: LLM-as-Judge is weaker than formal verification. Subject to model judgment errors.
- Strong tier false positive: Qwen generated correct max implementation but mislabeled it adversarial. Indicates need for formal false-positive filtering.
- Sampling bug: Statistical sampling run had a code error (wrong attribute name), preventing accurate multi-trial analysis. Qualitative patterns from output logs confirm consistency.
- No baseline comparison: Future work should compare against random code generation + verification as a lower bound.

7. Future Work

- Atlas IDE integration: Real-time spec validation as developers write specifications.
- Multi-verifier support: Extend Layer 1 to Lean 4, Coq, Isabelle, and F*.
- Benchmark scaling: Evaluate on DafnyBench (782 programs), VERINA, and Clover datasets.
- Automated strategy discovery: LLM-generated novel adversarial strategies beyond the four manual ones.

- Spec-repair fine-tuning: Use gap taxonomy as training data for automated specification fixing.
- Multi-model adversarial diversity: Different LLM families find different gaps.
- Formal false-positive filtering: Use proof search to verify adversarial intent, eliminating false positives like the strong-tier max case.
- CI/CD integration: Run SpecSaboteur on spec changes in pull requests.

8. Conclusion

SpecSaboteur demonstrates that adversarial implementation synthesis is a practical, effective, and novel approach to specification validation. The monotonic gradient across Dafny spec strength tiers validates both sensitivity and specificity. The iterative refinement loop achieves convergence in two iterations or fewer, showing that adversarial pressure systematically strengthens specifications. The extension to real-world software specifications demonstrates generalizability beyond formal methods. With a working two-layer pipeline, demonstrated convergence, and a catalog of gap patterns, SpecSaboteur is a concrete tool that moves the needle on ensuring that verified code is not just correct-by-construction, but correct-by-specification.

References

- [1] SpecGen: Automated Generation of Formal Program Specifications via LLMs. ICSE 2025.
- [2] SpecSyn: LLM-based Specification Synthesis and Refinement. 2026.
- [3] Self-Spec: Self-Supervised Model-Authored Specification Languages. OpenReview 2025.
- [4] Validating Specifications with LLM-Generated Test Cases. FM 2026.
- [5] adversarial-spec. github.com/zscole/adversarial-spec
- [6] Atlas formal-specification-ide. github.com/atlas-computing-org/formal-specification-ide
- [7] Atlas: AI and Formal Verification. atlascomputing.org/atlas-ai-and-formal-verification.pdf
- [8] Vericoding Benchmark: Formally Verified Program Synthesis. POPL 2026.