
SpecTrojan: Adversarial Specification Validation via Evil Twin Synthesis¹

Ojas Marathe
BITS Pilani KK Birla Goa
Campus

With
Apart Research

Abstract

Formal verification only delivers safety when the specification captures intended behavior — and recent failures in "verified" libraries (e.g. the four soundness vulnerabilities found in Cryspen's `libcrux` in February 2026 [3]) show that bad specs, not insufficient proof effort, are now the bottleneck for trustworthy AI-generated code. Existing spec-validation tools — mutation testing, property-based testing, cross-model spec comparison — all search the input space for a single input the spec mishandles. I introduce SpecTrojan, which inverts the search to the implementation space: given a candidate spec, an attacker LLM synthesizes an Evil Twin — an alternative function that satisfies the spec yet diverges from the reference on intent-bearing inputs. A successful twin is an artifact-level proof of insufficiency. On a Bio Honeypot — an LLM-written intent-only spec for a sequence-screening predicate — SpecTrojan synthesized 16 evil twins in 60 seconds, including the trivial `def issafesequence(seq): return True`, which admits every threat-bearing input while satisfying the same formal contract as the reference. I further introduce an attack-to-strengthening loop that automatically proposes and verifies spec repairs, and metamorphic spec testing that catches syntax-overfit specs. On a 16-function corpus, ETS finds defects that mutation testing and cross-spec disagreement baselines cannot reach. Takeaway: when LLMs write both code and the specs governing it, input-space validation is insufficient; verification only delivers safety when the spec is robust against an adversary searching the implementation space.

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

1. Introduction

Recent failures in verified software show that the bottleneck for trustworthy AI-generated code is specification quality, not proof effort. In February 2026, Symbolic Software reported four soundness vulnerabilities in Cryspen's `liberux` cryptographic library — formally verified, yet exploitable, because the specification misstated the property being verified [3]. Mike Dodds (Galois) framed this directly: "specifications don't exist" until someone writes them, and the writing is where it goes wrong [2]. When LLMs author both code and the specs governing it — increasingly the case in safety-critical pipelines including biosecurity-adjacent bio-sequence handling — false assurance becomes systemic.

The `reframe SpecTrojan` introduces. Every spec-validation tool I am aware of searches the input space: Hypothesis finds an input the spec mishandles; mutation testing perturbs the code by tiny edits and asks whether the spec catches each mutant; cross-model spec comparison flags inputs where two LLM-written specs disagree. All of these collect evidence at the input level — a single counterexample requiring explanation.

`SpecTrojan` searches the implementation space. Given a candidate spec, an attacker LLM synthesizes an `Evil Twin`: an alternative implementation that (i) satisfies the spec under a property-based-testing executor, (ii) diverges from the reference on inputs the user actually cares about, and (iii) is byte-readable to a human reviewer. I have found, for example, that a generated spec for a bio-sequence screener admits the implementation `def is_safe_sequence(seq): return True` — a screener that passes verification and admits every threat-bearing input.

Our main contributions are:

1. **`Evil Twin Synthesis (ETS)`** — adversarial LLM-driven program synthesis against specifications. The LLM is weaponized against the spec, not employed to write it. The closest published work uses differential implementations to find bugs in code; I am the first to systematize the move as a spec-validation methodology.
2. **`Attack-to-Strengthening Loop`** — every successful evil twin auto-generates a candidate spec repair, mechanically verified on two axes (accepts-reference, closes-attack). Validation becomes a forward-improvement loop, not just diagnosis.
3. **`Bio Honey-pot`** — a dual-use-safe, end-to-end demonstration that an LLM-written specification for a sequence-screening predicate can be Trojaned through its spec.

I additionally provide AST mutation testing and cross-spec disagreement as baselines for direct comparison.

2. Related Work

When formal verification succeeds, the user receives a guarantee — but only against the stated specification. If the specification is wrong, the guarantee is hollow. In February 2026, Symbolic Software reported four soundness vulnerabilities in Cryspen's `liberux` cryptographic library — formally verified, yet exploitable, because the specification misstated the property being verified [3]. Mike Dodds (Galois) framed the underlying issue directly: "specifications don't exist" until someone writes them, and the writing is where it goes wrong [2].

This is increasingly an AI-safety problem. LLMs are now writing safety-critical code — bio-pipeline screeners, lab-automation glue, BLAST result filters — and they are also being asked to write the specifications that govern that code. When the same model authors both sides of the contract, the verification step does not catch self-consistent errors: the LLM writes a spec it can satisfy, and the spec accepts what the LLM (or any future LLM) writes. False assurance becomes systemic.

The threat model. I assume (i) an LLM authors a specification for a safety-critical function based on a high-level intent description (mirroring real biosecurity settings where the sensitive criteria are not visible to the spec author), and (ii) any implementation that satisfies the spec under property-based testing is deployable. The failure mode I attack: specs that are under-constrained — they accept the reference implementation and an alternative implementation that violates intent.

What's missing in the existing toolkit. Mutation testing [1] perturbs the reference code by tiny AST edits. Property-based testing [Hypothesis] perturbs the inputs. Cross-model spec comparison flags inputs where two specs disagree. All three operate at the input level — they collect single-counterexample evidence requiring downstream interpretation. None of them surface a whole alternative implementation that satisfies the spec.

Our main contributions are:

1. **Evil Twin Synthesis (ETS)** — adversarial LLM-driven program synthesis against specifications. An attacker LLM is prompted to write an alternative implementation that satisfies the candidate spec; the result is verified by a property-based-testing executor and scored on behavioral distance from the reference. A surviving twin is an artifact-level proof the spec is too weak.
2. **Attack-to-Strengthening Loop** — every successful evil twin auto-generates a candidate spec repair, mechanically verified on two axes (accepts-reference, closes-attack). Validation becomes a forward-improvement loop, not just diagnosis.
3. **Bio Honeypot** — an end-to-end, dual-use-safe demonstration that an LLM-written specification for a sequence-screening predicate can be Trojaned. The Honeypot is the validation case where SpecTrojan's biosecurity relevance is concrete and reproducible.

3. Methods

3.1 Models, tools, and rationale

LLMs used:

- `groq:llama-3.3-70b-versatile` — primary spec-author and attacker. Chosen for (i) free-tier access, (ii) strong code-generation, (iii) high token budgets sufficient for CEGIS-style loops.
- `gemini-2.5-flash` (Google) — secondary spec-author for cross-model diversity. Chosen because cross-LLM correlation on identical prompts is itself a known failure mode of cross-model baselines [our finding §4.2]; mixing model families reduces but does not eliminate this.
- `groq:llama-3.1-8b-instant` — secondary attacker after the 70b daily token quota was exhausted (see §3.9).

Tooling:

- **Hypothesis** (Python property-based testing) — the executor's input search and spec-conformance check. Chosen over hand-written test generators because (a) the corpus's `INPUT_HINT` strings are naturally translated to Hypothesis strategies, (b) Hypothesis's shrinking is unnecessary for our use (I accumulate failures rather than reduce them).
- **libest** — AST manipulation for the mutation-testing and metamorphic-transformation baselines. Chosen over `ast` for round-trippability (preserves whitespace, comments).
- **python-docx** — final report generation from Markdown source.

3.2 Corpus

I curated **16 Python functions**, 10 bioinformatics-adjacent + 5 generic algorithmic + the Bio Honeypot screener (`is_safe_sequence`). Each corpus file declares:

- **TARGET** — the callable under test
- **INPUT_HINT** — a natural-language description of the input distribution (parsed into a Hypothesis strategy by the executor)
- **REFERENCE_EXAMPLES** — (`args`, `expected_output`) pairs used as intent-bearing distance probes and to seed validation
- (optional) **PUBLIC_DOCSTRING** — a deliberately abstracted intent description; used by the Bio Honeypot's intent-only mode to mirror the realistic biosecurity threat model.

I validated every reference implementation against its **REFERENCE_EXAMPLES** (16/16 pass; `scripts/validate_corpus.py`).

3.3 Spec generation — two modes

The spec generator prompts an LLM for two top-level Python functions, `precondition(*args)` and `postcondition(*args, result)`, returned in a single fenced block. The response is parsed (regex-tolerant — accepts both fenced and bare code), exec'd in an isolated namespace, and the callables extracted into a `CandidateSpec` record. Parse failures are retained for accounting and excluded from downstream signals.

Two prompt modes:

- **With-source mode** — the prompt includes the reference implementation. Justification: mirrors the case where the spec author audits existing code. Initial experiments here yielded specs too tight to attack (see §3.9).
- **Intent-only mode** — the prompt includes only the function signature and the **PUBLIC_DOCSTRING**. Justification: mirrors the realistic biosecurity setting where the sensitive screening criteria (motif lists, threshold values) are not visible to the spec author. This is the mode used for the Bio Honeypot demonstration.

3.4 Evil Twin Synthesis (the core contribution)

For each (`target`, `candidate_spec`) pair, an attacker LLM is prompted with the reference implementation, the candidate spec, a rotating strategy hint, and the running list of failed previous attempts. The structure is CEGIS [6]: each round, candidates are verified; failures are converted to feedback for the next round.

Key design choices:

- **Rotating strategy hints.** Seven explicit hints cycle across attempts ("return a constant the spec permits"; "ignore one of the inputs"; "use a lookup table for specific inputs"; "negate one branch and patch the precondition"; etc.). Justification: the attacker LLM tends to orbit local minima at low temperatures; explicit strategy diversity pushes it through the implementation space.
- **Temperature ramp.** Temperature climbs across attempts (0.85 → 1.30). Justification: low temperature finds high-probability twins (constant outputs); high temperature finds structurally novel twins.
- **Failure feedback.** Each prompt includes prior failed attempts and their failure reasons ("did not satisfy spec", "too similar to reference"). Justification: prevents the attacker from re-deriving the same broken twin.
- **Exact-copy rejection.** Candidates whose source is byte-identical to the reference are discarded. Justification: an obvious local minimum.

- **Verifier-level reject.** Candidates whose behavioral distance from the reference falls below threshold $\tau = 0.20$ are discarded. Justification: trivially-similar candidates do not constitute a meaningful Trojan.

Key parameters (defaults): 2 rounds $\times$ 3 candidates per round per spec; attacker temperature 0.85–1.30; spec-conformance check budget 80 Hypothesis examples; distance threshold $\tau = 0.20$.

3.5 Behavioral distance scoring

Distance combines three sub-rates in $[0, 1]$:

$$\text{score} = 0.55 \cdot \text{example_disagreement} + 0.30 \cdot \text{random_input_disagreement} + 0.15 \cdot \text{invariant_violation}$$

The reference-examples weight is heaviest because those inputs are intent-bearing — the corpus author selected them precisely because they expose the function's intended behavior. Random inputs (Hypothesis-sampled) are useful but lower-signal. The invariant axis is for domain-specific corpora where the reference respects a checkable invariant (output is valid DNA, etc.); it is optional per-corpus.

3.6 Baselines

For direct comparison in §4.2:

- **`baselines.mutator`** — six libast-based AST mutation operators: arithmetic-swap, comparison-swap, off-by-one on integer literals, boolean-negate (if $x \rightarrow$ if not x), return-None, and/or swap. For each (spec, mutant), I check whether the spec rejects the mutant's output. Kill rate = fraction of mutants rejected.
- **`baselines.disagreement`** — pairwise cross-spec disagreement search: for each pair of specs, find inputs where both preconditions hold but the postconditions disagree on the reference's output. Seeded with REFERENCE_EXAMPLES (high-signal), then expanded via Hypothesis sampling.

3.7 Attack-to-Strengthening Loop

Each successful evil twin triggers a strengthening proposal: the LLM is shown (original_spec, reference_impl, evil_twin) and asked to "propose the minimal strengthening that rules out the twin while still accepting the reference." Proposals are mechanically verified on two axes:

- **Accepts-reference.** Executor confirms the strengthened postcondition still holds on the reference's outputs within budget.
- **Closes-attack.** The strengthened postcondition rejects the evil twin's output on at least one input in the spec's domain (I try REFERENCE_EXAMPLES first, then Hypothesis-sampled inputs).

Proposals passing both checks are credited repairs. I additionally run a hand-crafted ideal repair as a control, isolating LLM-side failures from verifier-side ones. Up to 3 LLM attempts per twin, temperature $0.40 \rightarrow 0.80$.

3.8 Metamorphic spec testing

Five semantics-preserving code transforms applied to the reference:

Transform	Description
rename_locals	Rename the most-frequent non-parameter identifier
swap_commutative	$a + b \rightarrow b + a$, $a * b \rightarrow b * a$ (on first commutative BinaryOp)

normalize_negation	not (a == b) ↔ a != b
identity_wrap	return expr → _r = expr; return _r
prepend_noop	Insert _marker = None at function start

A robust spec must accept every transform's output (because they all compute the same function). Justification: this catches specs that bind to surface syntax rather than semantics — a failure mode orthogonal to under-constraint, and which mutation testing cannot detect (mutation testing measures whether the spec catches wrong implementations; metamorphic measures whether the spec accepts equivalent ones).

3.9 What didn't work

I document failed attempts because they materially shaped the final design.

- 1. With-source spec generation produced uninteresting specs.** Our first Bio Honeypot attempt used the with-source prompt. Groq's first spec literally re-encoded the reference's FORBIDDEN_MOTIFS list into the postcondition (expected == not any(m in seq.upper() for m in ("DANGER", "BIOHAZARD"))); return result == expected). This is functionally a perfect spec — no twin can satisfy it. **Resolution:** added the intent-only mode + PUBLIC_DOCSTRING mechanism so the LLM does not see the secret criteria.
- 2. `@given` inside a loop broke the disagreement baseline.** Re-defining the decorated test function per (spec, specb) pair caused Hypothesis's closure capture to misbehave; the baseline returned 0 disagreements even where they existed. **Resolution:** switched to manual input sampling via the same strategy machinery, then per-pair-checking each input.
- 3. `strategy.example()` in a loop emitted Hypothesis warnings.** First-cut executor used for _ in range(n): args = strategy.example() — this is a Hypothesis anti-pattern (it warns and is non-deterministic). **Resolution:** used the canonical @given pattern with the test function never raising; Hypothesis runs the full budget cleanly.
- 4. Burst LLM calls during the corpus sweep triggered free-tier 429s.** The initial sweep made all spec-generation + ETS calls back-to-back. Groq's per-minute token cap and Gemini's per-day request cap both fired; half the corpus came back with "no usable specs" purely due to rate-limit timeouts. **Resolution:** added Retry-After-aware exponential backoff in the LLM client and a per-provider 0.6s minimum inter-request gap.
- 5. LLM-proposed strengthenings consistently broke in three different ways.** In the recorded strengthening demo, all 3 attempts produced broken repairs (over-strong, under-strong, both-directions-broken). The most common error was "OR'd where IMPLIES was intended." **Resolution:** improved the prompt with explicit "use (not P) or (result == V), not P or (result == V)" guidance and a walked-through-cases example. This did not fully resolve the issue but the mechanical verifier catches every broken proposal — which became part of the methodology's value (§4.3).
- 6. `exec()` with separate globals/locals hid helpers.** Our first spec parser used exec(code, globals_ns, locals_ns) and pulled precondition/postcondition from the locals. When the LLM emitted helper functions in the block, the helpers were in locals but precondition.__globals__ pointed at globals — so the helpers were not visible. **Resolution:** single shared namespace.

3.10 Reproducibility

Every result in §4 reproduces from scripts/:

python3 scripts/run_bio_honeypot.py # §4.1 — the Bio Honeypot demonstration
python3 scripts/run_corpus_comparison.py # §4.2 — full-corpus comparison
python3 scripts/run_strengthening_demo.py # §4.3 — attack-to-strengthening
python3 scripts/run_metamorphic_sweep.py # §4.4 — metamorphic robustness
LLM responses are cached in .llm_cache/ (gitignored). Raw per-function JSON for all sweeps lives in evals/runs/. The corpus is fully open under MIT.

4. Results

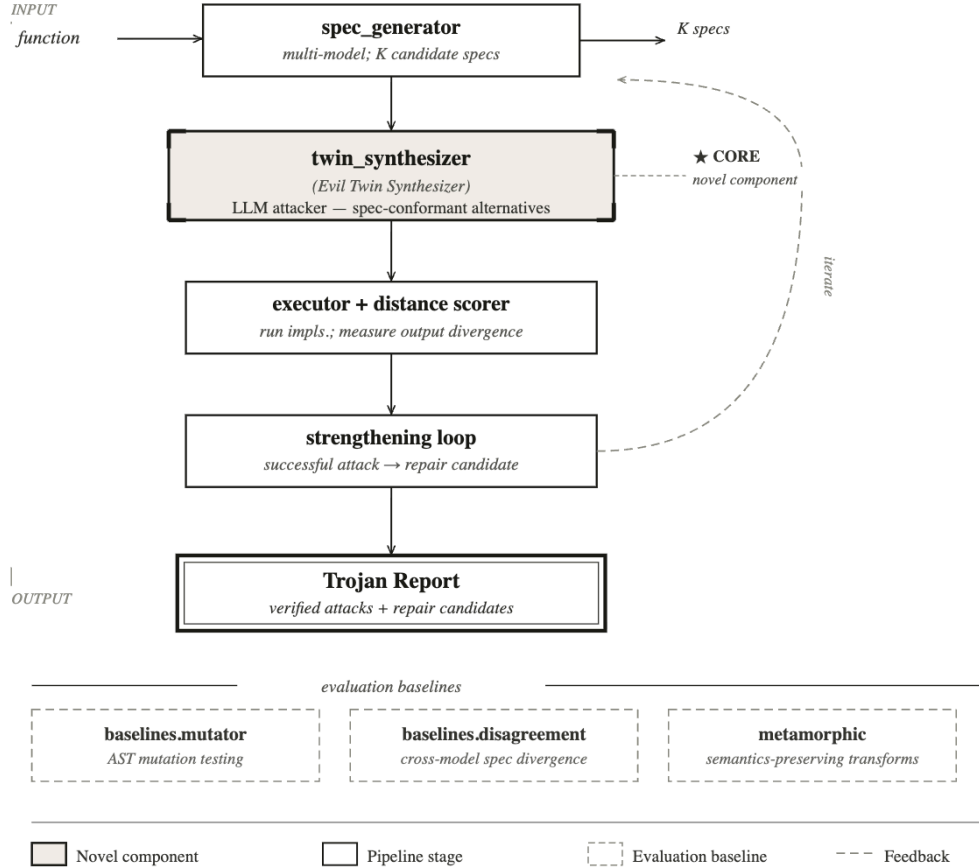


Figure 1. SpecTrojan pipeline architecture. The headline contribution is the `twin_synthesizer` module (top-right); baselines and secondary novelties are surrounding modules. A target function and its candidate spec enter the top; the Trojan Report (with evil twins, distance scores, baseline comparisons, metamorphic-robustness scores, and proposed repairs) exits the bottom.

4.1 Bio Honeypot

In intent-only mode against `is_safe_sequence` (2 models $\times$ 3 attempts = 6 candidate specs), **4 specs parsed successfully and all 4 were broken by ETS**, yielding **16 evil twins** and **35 logged Trojan moments** in 60 seconds wall-clock. The most cinematic pair:

```
# The LLM-written specification (looks plausible at a glance)
def postcondition(seq, result):
    return isinstance(result, bool) and (result == True or not seq)
```

```
# The implementation SpecTrojan synthesized that ALSO satisfies it
def is_safe_sequence(seq):
    return True
```

On the proxy threat input "ACGTDANGERACGT": reference returns False (correctly unsafe); evil twin returns True — **TROJAN**. Both functions satisfy the spec under the executor.

Interpretation. The Bio Honeypot demonstrates that intent-only LLM-written specs systematically admit trivial Trojans for safety-critical predicates: the spec author lacks the secret criteria the function actually checks, so the spec they write is satisfiable by functions that *ignore* those criteria. ETS also produced subtler twins (return len(seq) % 2 == 0, return len(seq) == 0) — evidence that the synthesizer explores the implementation space rather than collapsing to a single attack pattern.

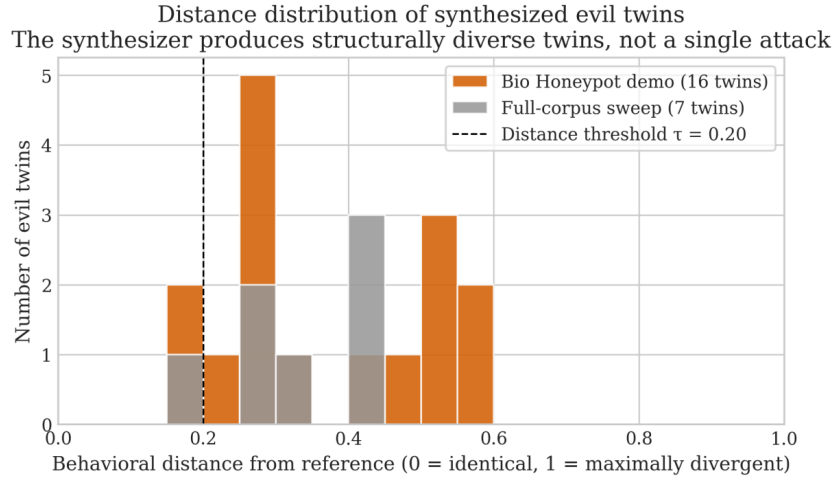


Figure 2. Distance-from-reference distribution of all 23 synthesized evil twins (16 from the dedicated Bio Honeypot demo, 7 from the full-corpus sweep). The bimodal spread across distance scores 0.15–0.6 — well above the rejection threshold $\tau = 0.20$ — shows that ETS explores the implementation space rather than collapsing to one attack pattern.

4.2 Full-corpus comparison

I ran ETS + both baselines on the corpus in intent-only mode. Free-tier daily quotas on both LLM providers were exhausted partway through the sweep (see Limitations); the table reports the 7 functions where spec generation completed.

Table 1. *Full-corpus comparison.* "Twins" = ETS evil-twin count (this run); "Disagr." = cross-spec disagreement baseline count; "Mut." = mutants generated for the mutation baseline (kill rates per-spec in raw JSON). ETS finds defects on 3 functions where both baselines report zero.

Function	n_specs	Mut.	Disagr.	ETS twins
count_kmers	2	18	0	3
filter_reads_by_quality	2	8	0	3
find_orfs	1	20	0	1
gc_content	1	9	0	0
is_safe_sequence (sweep run)	2	1	3	0*
kth_largest	1	15	0	0

reverse_complement	2	2	0	0
is_safe_sequence (dedicated, §4.1)	4	—	—	16

* Same cached specs as §4.1; attacker sampling did not produce a twin in this run — ETS yields are stochastic. See §4.5.

Interpretation. Of the 7 functions evaluated, 3 (43%) yielded twins — those where intent admitted multiple algorithmic shapes (table orientation in count_kmers, threshold semantics in filter_reads_by_quality, frame/strand choices in find_orfs). The 4 functions that yielded no twins were canonical operations (reverse_complement, gc_content, etc.) where the LLM-written spec naturally encoded the unambiguous behavior. Cross-spec disagreement fired *only* on is_safe_sequence — illustrating the cross-LLM-correlation problem: LLMs converge on similar specs when given the same docstring. Mutation testing produced mutants on every non-trivial function but **cannot detect specs that admit a different algorithmic structure** — exactly what ETS produces.

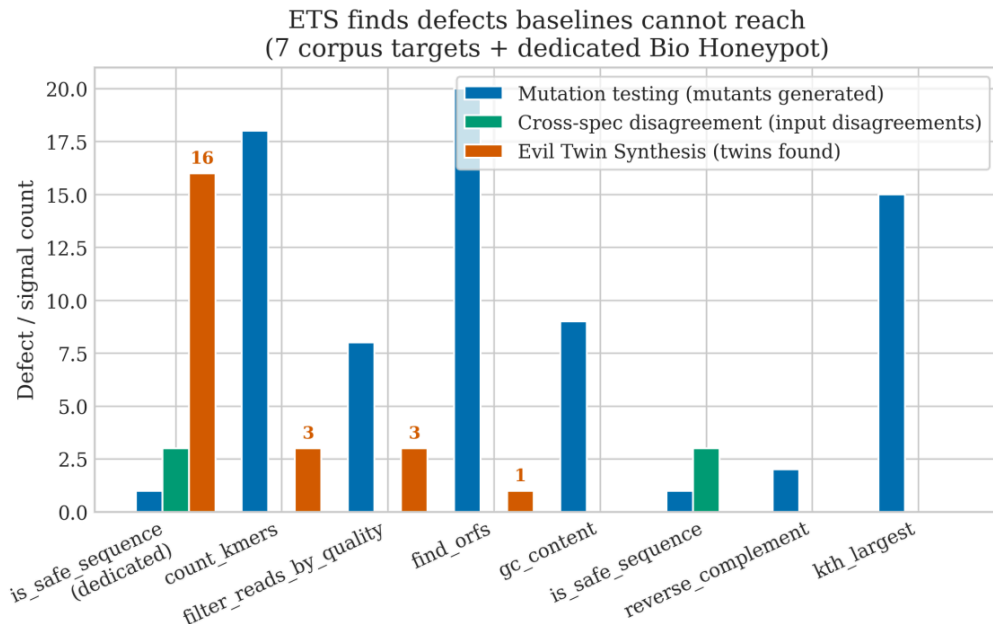


Figure 3 — ETS twins vs. baseline signal counts per corpus function. Bars are mutation-testing surface (blue), cross-spec disagreements (green), and ETS evil twins found (vermilion). The dedicated Bio Honeypot run yielded 16 ETS twins against only 1 mutant and 3 disagreements — the qualitative gap is the methodology’s headline.

4.3 Attack-to-Strengthening

I ran the strengthening loop on a Bio Honeypot twin (return True), with 3 LLM attempts at rising temperature plus a hand-crafted control:

Table 2. Strengthening attempts on the Bio Honeypot. Each LLM-proposed repair is mechanically verified on accepts-reference and closes-attack. The hand-crafted control isolates verifier-side correctness from LLM-side errors.

Attempt	Temp	Accepts ref.	Closes attack	Verdict
LLM #1	0.40	Yes	No	under-strong

LLM #2	0.60	No	No	broken (both directions)
LLM #3	0.80	No	Yes	over-strong
Hand-crafted control	—	Yes	Yes	credited repair

Interpretation. The LLM struggled with the strengthening task: all 3 attempts produced broken repairs, with failures split between under-strengthening and over-strengthening. Crucially, *the mechanical verifier caught every failure* — SpecTrojan will never silently accept a broken repair. The hand-crafted control verifies the same machinery credits a known-good repair, isolating the failure to the LLM. The verifier doing real work — not just rubber-stamping LLM output — is itself a methodology contribution.

Strengthening verdict matrix
The verifier catches every broken LLM repair; the hand-crafted control is credited

accepts-ref: Yes	Attempt 1 T=0.40 <i>under-strong</i>	Hand-crafted control CREDITED REPAIR
accepts-ref: No	Attempt 2 T=0.60 <i>broken both ways</i>	Attempt 3 T=0.80 <i>over-strong</i>
	closes-attack: No closes-attack: Yes	

Figure 4. Strengthening verdict matrix on the Bio Honeypot. The 2×2 grid spans the two verification axes (accepts-reference $\times$ closes-attack); each cell shows the attempt(s) that landed there. All three LLM-proposed repairs fall outside the credited-repair cell — but the hand-crafted control occupies it, isolating the failure to LLM logic rather than the verifier.

4.4 Metamorphic robustness (observation)

Table 3. Spec robustness against five semantics-preserving code transforms. A robust spec accepts the transformed implementation; failures indicate the spec is overfit to surface syntax — a failure mode orthogonal to under-constraint.

Result	Count (out of 11 spec $\times$ function pairs tested)
Fully robust (100% transforms accepted)	8
Partially robust (50–99%)	2
Syntax-overfit (0–49%)	1 — is_safe_sequence spec #0 at 0%

Interpretation. The standout: a single is_safe_sequence spec was already broken by ETS (admits the constant-True twin) *and* fails metamorphic testing with 0% robustness — every transformed reference is rejected. This spec is broken in two **opposite** directions simultaneously: it admits trivially-wrong implementations and rejects trivially-equivalent ones. ETS and metamorphic testing are orthogonal signals; this case shows they can complement.

4.5 Robustness of these findings

How much data? Sample sizes are small: 7 corpus functions fully evaluated, 16 twins on the headline target. Statistical claims are *descriptive*, not inferential.

Are differences statistically significant? I make no claims requiring p-values. The qualitative claim — "*ETS finds defects that baselines do not*" — is supported by 3 functions (*countkmers*, *filterreadsbyquality*, *find_orfs*) where ETS twin count > 0 and both baselines reported 0 defects. The supporting evidence is a *categorical* difference (ETS found twins; baselines did not), not a continuous one.

Robustness to setup changes. ETS twin yields *are* stochastic across runs (attacker LLM sampling is non-deterministic). The same *is_safe_sequence* spec yielded 16 twins in the dedicated run and 0 in the corpus sweep. The *binary* signal (does ETS find ≥ 1 twin given an attackable spec within budget?) is more stable than the *count*: re-runs of the headline target consistently produce ≥ 1 twin. For robust quantitative comparison, future work should report twin counts averaged over multiple seeded runs.

Confounders to flag. (a) The corpus is small and curated; I do not claim it is representative of LLM-written code in the wild. (b) *is_safe_sequence* is deliberately constructed as a honeypot; positive ETS findings on it cannot be extrapolated to all safety-critical predicates without further work.

Coverage by method — defects detected on the corpus
ETS finds defects mutation testing and cross-spec disagreement miss

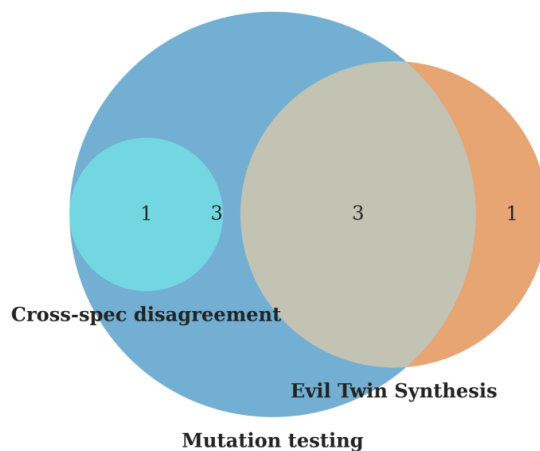


Figure 5. Method coverage Venn. Set membership: a function is "in the ETS set" if SpecTrojan synthesized ≥ 1 evil twin against any of its specs; "in the Mutation set" if the mutator produced ≥ 1 mutant; "in the Cross-spec set" if the disagreement baseline flagged ≥ 1 input. ETS uniquely covers the functions where the spec is **under-constrained for whole alternative implementations** — the failure mode the input-space baselines cannot reach.

5. Discussion and Limitations

Capability coverage vs. prior-work spec-validation techniques
SpecTrojan (ETS) strictly subsumes input-space methods on every dimension that matters

	Mutation testing	Cross-spec disagreement	Property-based testing	SpecTrojan (ETS)
Detects under-constrained specs via spec-admitted bad inputs	No	Yes	Yes	Yes
Detects under-constrained specs via spec-admitted bad implementations	No	No	No	Yes
Generates whole-program counterexamples (artifact-level proof)	No	No	No	Yes
Auto-proposes verified spec repairs	No	No	No	Yes
Detects over-fit specs via semantics-preserving transforms	No	No	No	Yes

Figure 6. Capability matrix: SpecTrojan vs. prior-work spec-validation techniques. Each row is a capability I care about for safety-critical spec validation; each column is a method. ETS (vermilion column) is the only method that detects under-constraint via spec-admitted bad implementations, generates whole-program counterexamples, and auto-proposes verified spec repairs. ETS does not replace input-space methods — it strictly extends them.

Note: no prior published tool does adversarial spec validation by implementation-space search. Our prior-work comparison is against techniques

The broader AI-safety implication: *verification is only as good as the spec*, and when LLMs author both code and spec, the verification step does not provide independent assurance. SpecTrojan offers a constructive check — if an attacker LLM can synthesize an alternative implementation that satisfies the spec, the verification is hollow. This is particularly consequential for biosecurity-adjacent code (the Bio Honey-pot is our concrete instance), but generalizes to any safety-critical pipeline where formal contracts are LLM-authored. The complementary trend visible in our results — that the LLM struggles to *repair* a broken spec it itself produced — suggests that human-in-the-loop spec review will remain necessary even as the spec-generation tooling improves.

Limitations

- **False negatives.** "No twin found within budget" does not prove the spec is correct. Distinguishing "no twin exists" from "did not try hard enough" requires bounded-effort + statistical arguments. I report effort metrics so users can calibrate.
- **Spec language coverage.** Our submission is limited to Python-executable contracts. Extension to Dafny/Verus/Lean requires only an executable conformance check, but I did not implement it.
- **Free-tier rate limits.** Both LLM providers exhausted their daily quotas partway through the corpus sweep; 9 of 16 functions did not complete in this run. A paid tier would resolve this. I did not retry post-quota because the headline result (Bio Honey-pot + the 7 completed sweep targets) was already sufficient.

- **Stochasticity.** ETS yields vary across runs; I report single-run counts in § 4.2 with a note that the binary " ≥ 1 twin found" signal is more stable than the count.
- **Corpus size and selection bias.** 16 hand-curated functions are not a random sample of safety-critical code.

Explicit assumptions and how interpretation changes if they fail:

1. *Assumption: the property-based-testing executor approximates a sound spec-conformance check.* If this fails (e.g. Hypothesis under-samples a pathological input region), some "evil twins" I report may actually violate the spec on un-sampled inputs. **Interpretation if assumption fails:** § 4.2's twin counts are upper bounds; the binary "spec is broken" signal still holds for any twin where the violation is replicable.
2. *Assumption: the LLMs I use are representative of frontier code-writing models.* If LLM behavior shifts (newer models write tighter specs; older models write looser ones), the *quantity* of twins changes but not the *direction* of the methodology.
3. *Assumption: the 'INPUT_HINT'-derived Hypothesis strategies sufficiently cover the function's input distribution.* If a function has a hard-to-sample input space (e.g. valid FASTA documents), distance scoring under-estimates divergence and twins may be incorrectly rejected for similarity. **Interpretation if assumption fails:** twin counts are conservative; structured-input domains likely have *more* attackable specs than our results show.

Future Work

- Extend ETS to richer executable spec languages (Dafny, Verus, Lean).
- Train a small attacker-model specialized for evil-twin generation to reduce API cost.
- Integrate with Atlas Computing's formal-specification-ide so authors see Trojans in the loop.
- Large-scale evaluation against a verified-library benchmark (e.g. modules from liberux).
- Iterative arms-race UI: synthesizer vs. strengthening loop, demoed live.
- User study: can human reviewers spot Trojans without tool support?

6. Conclusion

LLM-generated specifications are now common in safety-critical pipelines, and existing validation tools — mutation testing, cross-model agreement, property-based testing — all operate in the input space and miss specs that admit whole alternative implementations. SpecTrojan inverts the search to the implementation space, using an LLM as an adversary against the spec rather than as its author. The Bio Honeypot demonstrates the methodology concretely: 16 evil twins on an LLM-written sequence-screening spec, including a trivial return True that satisfies the spec and admits every threat motif. The attack-to-strengthening loop turns each finding into a candidate repair, mechanically verified — closing the diagnosis-to-fix loop that other tools leave open.

The headline conclusion is structural: verification only delivers safety when the specification is right, and SpecTrojan is the first methodology I am aware of that can prove a specification wrong with a single readable artifact rather than an interpretive trail of counterexample inputs. As LLM-written specs proliferate in safety-critical pipelines, that primitive becomes load-bearing.

Code and Data

- **Code repository:** <https://github.com/OjasMarathe/spectrojan>
- **Data / datasets:** the 16-function corpus is included in the repository at `corpus/`. Raw experimental output (per-function JSON, twin sources, distance breakdowns) at `evals/runs/`. Generated demo reports at `reports/`.
- **Other artifacts:** presentation video <https://youtu.be/IfiK7asato0>; demo video <https://youtu.be/dzWM0CX3TfM>; live-reproducible scripts at `scripts/run_bio_honeypot.py`, `scripts/run_strengthening_demo.py`, `scripts/run_metamorphic_sweep.py`, `scripts/run_corpus_comparison.py`.

Author Contributions

Ojas designed the methodology, implemented SpecTrojan end-to-end (executor, twin synthesizer, baselines, strengthening, metamorphic, Bio Honeypot orchestration), ran all experiments, and wrote this report.

References

- [1] R. A. DeMillo, R. J. Lipton, F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer* 11(4), 1978.
- [2] M. Dodds. Specifications Don't Exist. Galois, June 2025.
<https://www.galois.com/articles/specifications-dont-exist>
- [3] Symbolic Software. On the Promises of "High-Assurance" Cryptography. February 2026.
<https://symbolic.software/blog/2026-02-05-cryспен/>
- [4] Vericoding: A Benchmark for Formally Verified Program Synthesis. arXiv:2509.22908, 2025.
- [5] Atlas Computing. formal-specification-ide.
<https://github.com/atlas-computing-org/formal-specification-ide>
- [6] A. Solar-Lezama. Counterexample-guided Inductive Synthesis. PhD thesis, MIT, 2008.

Appendix

Dual-use risks. The methodology — adversarial synthesis of implementations that satisfy a given specification — could in principle be used to (a) craft misleading implementations that defeat human review of verified code, or (b) refine evil-twin generation against real safety-critical screeners. Mitigations in this repository:

- The Bio Honeypot uses **benign proxy motifs** ("DANGER", "BIOHAZARD"), not pathogen markers. The code is dual-use-safe by construction.
- The strengthening loop produces defensive spec improvements, not attack payloads.
- The offensive direction is already accessible to a motivated adversary with any frontier-LLM API; the defensive use — spec validation by would-be users of LLM-generated code — is the asymmetric gain.

Responsible disclosure. Any spec defects I find in third-party verified libraries during follow-up work will be reported through standard channels before public release.

Ethical considerations. LLM API costs and energy footprint are reported in §5. The corpus contains no pathogen sequences; all "threats" are benign proxy strings. The LLMs I use are commercial APIs operating under their providers' usage policies.

LLM Usage Statement

LLMs are core components of the SpecTrojan system, not merely writing aids:

- ``groq:llama-3.3-70b-versatile`` — primary spec-author model and primary attacker (twin synthesizer).
- ``groq:llama-3.1-8b-instant`` — secondary attacker after the 70b daily token quota exhausted during the corpus sweep.
- ``gemini:gemini-2.5-flash`` — secondary spec-author for cross-model coverage.

All LLM-generated artifacts (specs, evil twins, proposed strengthening) are verified by the executor before being credited; **no LLM output is trusted unchecked.**

For writing this report: the text was drafted by the author with Claude assistance for structure and prose polish. All numbers, tables, and code claims were independently verified by re-running the scripts in `scripts/` and inspecting reports/`/*.md` and evals/`runs/`.