
Counterexample-Guided Validation & Repair of LLM-Generated Safety Specifications

Yifei Lu
Independent

With
Apart Research

Abstract

LLMs can turn natural-language safety requirements into formal specifications, but a fluent specification can still be the wrong one. It might quietly drop a guard, block access the policy should allow, or respond inconsistently when a safety-relevant input changes. We built a pipeline that checks whether an LLM-generated specification matches the intended policy, using biosecurity access control as a testbed. Each requirement is broken into traceable safety obligations such as clearance, training, approval validity, expiration, external anonymization, and emergency logging. Candidate specifications are written in a restricted DSL, and four oracles judge each one together: obligation coverage, sampled behavioral agreement, metamorphic relations, and SMT-guided counterexample search over Z3. Every candidate ends up with a risk-weighted review card. Across 30 simulated LLM candidates and 60 controlled mutants, small manual test suites caught only 63.3% of injected faults and 55.3% of the high-severity ones, while systematic validation reached 98.9% overall and 100% on high-severity faults. Feeding counterexamples back as repair signals fixed 29 of 30 candidates, raising the mean score from 70.1 to 95.4 and clearing every high-severity under-constraint, with one regression. In a live run, specifications from Gemini 3.5 Flash were parsed and validated end-to-end, and obligation-guided prompting reached full obligation coverage. Together, these results show that LLM-generated safety specifications can be validated systematically, with each failure traced back to a specific missing guard.

1. Introduction

LLMs are now used to translate natural-language requirements into formal artifacts, including access-control specifications that determine who may see what. This makes a hard expert task fast and accessible, but it creates a new problem. A specification that reads well and parses cleanly is not always the specification the requirement asked for. The model can produce something fluent and wrong at the same time, and because the output looks authoritative, the error is easy to miss. The autoformalization literature documents this directly, noting that models translating informal statements into formal logic are prone to hallucinated output and to the semantic shift between natural language and formal logic, so their formalizations can be syntactically valid yet logically inconsistent or simply wrong through unintended insertions, omissions, and misinterpretations [1, 2].

Safety-relevant access control is a sharp instance of this problem, because one dropped clause can open an unintended path. Access-control misconfigurations are among the main causes of security and privacy breaches, and the core difficulty is one of faithfulness, since it is hard to identify discrepancies between a policy specification and the intended function it is meant to enforce [3]. This issue is amplified when the author of the policy is a language model rather than a human engineer.

We target three concrete failure modes. A specification can be under-constrained, allowing access the intended policy forbids, which is the most dangerous case because a guard has been lost. It can be over-constrained, denying access the policy permits, which is less unsafe but signals an unfaithful translation. Or it can be locally inconsistent, responding differently to two inputs that differ only along a safety-relevant axis such as record risk or approval expiry. Our threat model assumes the model is cooperative rather than adversarial, so the errors come from ordinary misreadings of natural language, which is what makes them realistic and easy to overlook.

This class of error is checkable. Formal methods already verify human-written access policies at production scale. AWS encodes policies into SMT and checks millions of them daily to ensure storage buckets are not unintentionally public [4], and recent work models access policies in first-order logic to exhaustively search for counterexamples to security invariants, returning a concrete request that witnesses each violation [5]. We ask whether the same machinery can be turned on LLM-authored specifications, and whether it can do more than flag a problem by also explaining it. In practice, this means a reviewer approving a model-drafted access policy could run it through the pipeline and receive a ranked list of unsafe cases, each with the missing guard named and a concrete input that breaks it, instead of trusting that fluent output is correct.

Our main contributions are:

1. An end-to-end validation pipeline that combines obligation coverage, sampled behavioral agreement, metamorphic relations, and SMT counterexample search into a single risk-weighted review card, and a controlled benchmark of 30 LLM-style candidates and 60 fault-injected mutants on which to measure it.
2. A quantitative comparison showing that small manual test suites miss roughly a third of injected faults and nearly half of the high-severity ones, while systematic validation detects 98.9 percent overall and 100 percent of high-severity faults, and a demonstration that the multi-oracle's main added value is diagnostic rather than detective.
3. A counterexample-guided repair loop that fixes 29 of 30 faulty candidates and clears every high-severity under-constraint, plus a live evaluation on Gemini 3.5 Flash showing that obligation-guided prompting yields full obligation coverage.

2. Related Work

The closest line of work uses LLM to translate natural language into formal specifications and then checks the result against a checker: PAT-Agent generates model-checker inputs and runs a verification-guided repair loop [1], and related systems target verifiable C code, hardware description languages, and theorem-prover statements [6, 7]. These systems mostly ask whether an artifact passes a verifier. Our question is about faithfulness to a stated policy, so instead of returning a single pass or fail, we score the specific ways a specification departs from the intended behaviour.

A second body of work verifies access policies with SMT solvers, and it is mature for human-authored policies. AWS Zelkova and Block Public Access encode policies into SMT and check policy properties at the scale of millions of policies a day [4, 8], and first-order-logic encodings detect conflicts, unreachable denies, and unintended permissions before a policy is deployed [5, 9]. Our SMT oracle reuses this idea of a solver-generated counterexample. The difference is that we apply it to machine-generated specifications and pair every counterexample with obligation tracing, so each one maps to a named missing guard rather than an anonymous logical clause.

We also draw on two testing traditions. Metamorphic testing addresses the oracle problem by checking necessary relations between a source input and a transformed follow-up input, so a violation reveals a fault even when a full reference oracle is unavailable [10].

Counterexample-guided inductive synthesis alternates between a candidate and a verifier that returns a counterexample to refine it [11, 12], and recent work drives that loop with a language model [13]. We combine the two. Metamorphic relations catch inconsistencies that sampling alone can leave hidden, and our repair stage treats each disagreement as a counterexample-guided signal.

The missing piece across all of this is that prior tools either verify human-written policies or check whether a model-generated artifact compiles. We take an LLM-generated safety specification, rank its failures by severity, trace each failure to a specific obligation, and feed the counterexamples back to repair it inside one pipeline. Someone would reach for our method when the author of a safety policy is an LLM, and the cost of a silently dropped guard is high, because a passing verifier or a clean parse says nothing about whether a missing clause has quietly widened access. The insight it adds is a severity-ranked, obligation-level account of why a generated specification is unfaithful, together with a concrete input that proves it, which is the information a reviewer needs to decide whether to trust the policy and how to fix it.

3. Methods

We use biosecurity access control as a testbed because the stakes of an over-permissive policy are easy to motivate and the decision is a clean Boolean. Each input has three parts. A user carries a role (PI, researcher, trainee, or external), a clearance level from 0 to 5, and four flags for training completed, approval active, approval expired, and supervision by a PI. A record carries a risk level from 0 to 5 and flags for anonymized and sensitive. A context carries flags for emergency and logged. Enumerating these fields gives a finite space of 36,864 distinct inputs per requirement. For behavioural testing, we draw a reproducible sample of 2,000 inputs per requirement under a fixed seed, with 900 of them drawn from a boundary pool of extreme clearance or risk values and external or trainee roles. This boundary weighting follows standard boundary-value testing practice, since policies tend to break at the edges of their numeric ranges.

We wrote 10 natural-language requirements and, for each, a gold specification that serves as the behavioural reference and a set of named safety obligations with severities. The obligations cover clearance, training, approval validity, expiration, external anonymization, and emergency logging. Candidate specifications are written in a restricted DSL whose only operators are AND, OR, NOT, the comparisons EQ, NEQ, GE, GT, LE, and LT, and the leaf forms FIELD and CONST. The restriction is a deliberate design choice. A specification in this DSL is both executable on sampled inputs and translatable into SMT constraints, which is what lets every oracle reason about the same object. This mirrors how production policy verifiers encode access policies into a solver-friendly logic before checking them [4, 5].

To measure the validator against ground truth we built two controlled sets of test objects. The first is 30 LLM-style candidates, three per requirement, that simulate plausible generation mistakes such as missing obligations, wrong comparators, wrong Boolean connectives, unnecessary restrictions, and omitted emergency logging. The second is 60 mutants, six per requirement, produced by automatically injecting known faults into the gold specifications through dropping obligations, flipping comparators, replacing AND with OR, and negating Boolean conditions.

Because the injected fault is known, these objects measure detection against a labeled answer in the way mutation testing does [14].

The core of the method is four oracles applied to every candidate. The obligation coverage oracle compares the atomic predicates implied by each obligation with those present in the candidate and flags missing obligations and high-severity gaps. The behavioral agreement oracle runs the candidate and the gold specification on the 2,000 sampled inputs and separates disagreements into under-constrained cases, where the candidate allows access that gold denies, and over-constrained cases, where the candidate denies access that gold allows. The metamorphic oracle applies five safety-monotonic relations, for example raising a record's risk level should never turn a deny into an allow, and counts how often a relation is violated [10]. The SMT oracle encodes both specifications into Z3 over the finite domain and searches for an input on which they disagree, returning that input as a concrete counterexample.

The four oracles merge into a single review card with a score from 0 to 100. The score is 25 times the coverage rate, plus 25 times the agreement rate, plus 20 times the metamorphic consistency, plus 20 times an indicator that no high-severity under-constraint is present, plus 10 times an indicator of SMT consistency. A candidate is labeled high risk if it misses a high-severity obligation or allows unsafe access, medium if it disagrees anywhere, and low otherwise. The weighting places the most points on the failures that correspond to a lost guard, so the score tracks safety rather than surface similarity.

To test whether counterexamples carry a usable repair signal, we ran a controlled repair loop in the style of counterexample-guided inductive synthesis [11, 12]. Missing obligations are added back into the candidate, and recognized over-constrained patterns are replaced with the gold reference, after which each repaired candidate is re-scored by the same validator. We kept this stage as a controlled simulation rather than a live model loop so that the repair signal could be measured in isolation [13].

Finally, we evaluated live generation. We used Gemini 3.5 Flash through the Gemini API to produce DSL specifications directly from the requirements under three prompt modes. Direct translation simply asks for the specification, obligation-guided supplies the named obligations in the prompt, and adversarial review asks the model to critique and tighten its own draft. The model returns JSON in the same DSL, which is parsed and run through the identical validator.

Two parts of the live setup did not work as first intended, and we report them plainly. We first requested Gemini structured output with a response schema, but this was not reliable across calls, so the pipeline falls back to ordinary JSON prompting and standard parsing when structured output is unavailable. The free-tier API also rate-limited us, and 10 of the 30 generation attempts returned quota errors rather than specifications, which removed three requirements from the live results. On the repair side, adding obligations back fixed missing-guard faults completely but left

some comparator-level faults only partially corrected, so a few repaired candidates plateaued in the high 80s rather than reaching a full score.

The entire pipeline runs from a single notebook with a fixed random seed, the gold specifications and obligations are defined in code, and the API key is never stored or printed, so the controlled experiment is fully reproducible and the live experiment is reproducible up to the model provider's availability

4. Results

Manual testing is not enough, as shown a six-case hand-picked boundary suite, which detected only 63.3 percent of injected faults and 55.3 percent of high-severity faults. Every systematic strategy, random sampling, property-weighted sampling, and the full multi-oracle reached 98.9 percent overall and 100 percent on high-severity faults (Figure 1). The gap between manual and systematic is the headline. A small fixed test set, the kind a developer writes by hand, lets roughly one in three faults and almost half of the dangerous ones through.

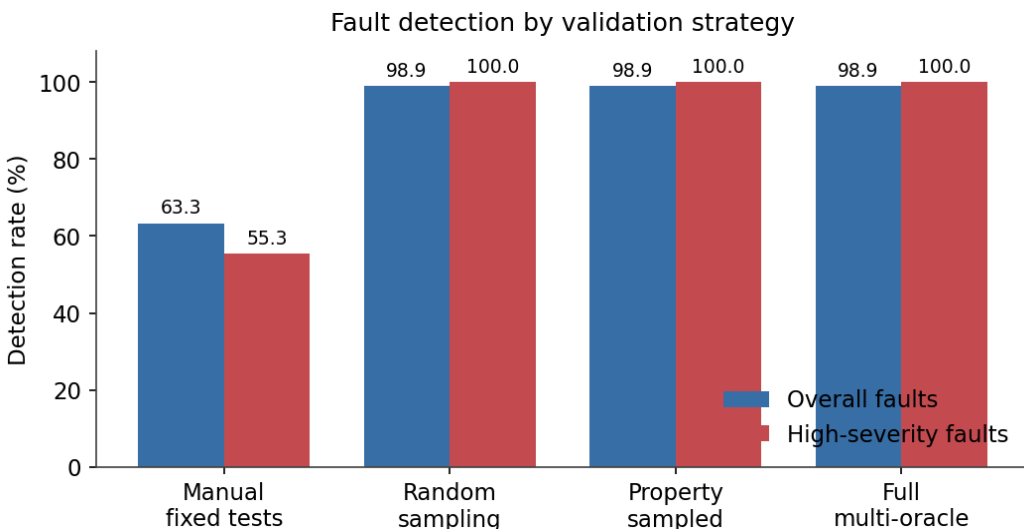


Figure 1. Fault detection rate by validation strategy across 90 controlled objects. Manual testing detects 63.3 percent of faults (55.3 percent of high-severity), while all systematic strategies reach 98.9 percent and 100 percent. Higher is better.

The multi-oracle's value is diagnosis; all three systematic strategies land on the same 98.9 and 100 percent, which is itself a finding. Sampling alone already catches almost everything behavioral. The reason to run the full multi-oracle is not a higher catch rate but what it tells you: which named obligation is missing, a concrete SMT counterexample, a severity label, and a repair

target. In our runs, the metamorphic oracle also caught cases that pure sampling can miss, since a relation can expose an inconsistent response to a controlled input change even when overall agreement looks high.

Counterexamples convert into repairs. Feeding disagreements back as repair signals fixed 29 of 30 candidates (96.7 percent), with a single regression and one already-correct candidate left unchanged. Mean validator score rose from 70.1 to 95.4, mean disagreements per candidate fell from 346.9 to 50.8, and the count of high-severity under-constrained candidates dropped to zero (Figure 2). The remaining gap to a perfect 100 is mostly comparator-level mismatches that our controlled repair action did not target, which is honest evidence that adding obligations back is necessary but not always sufficient.

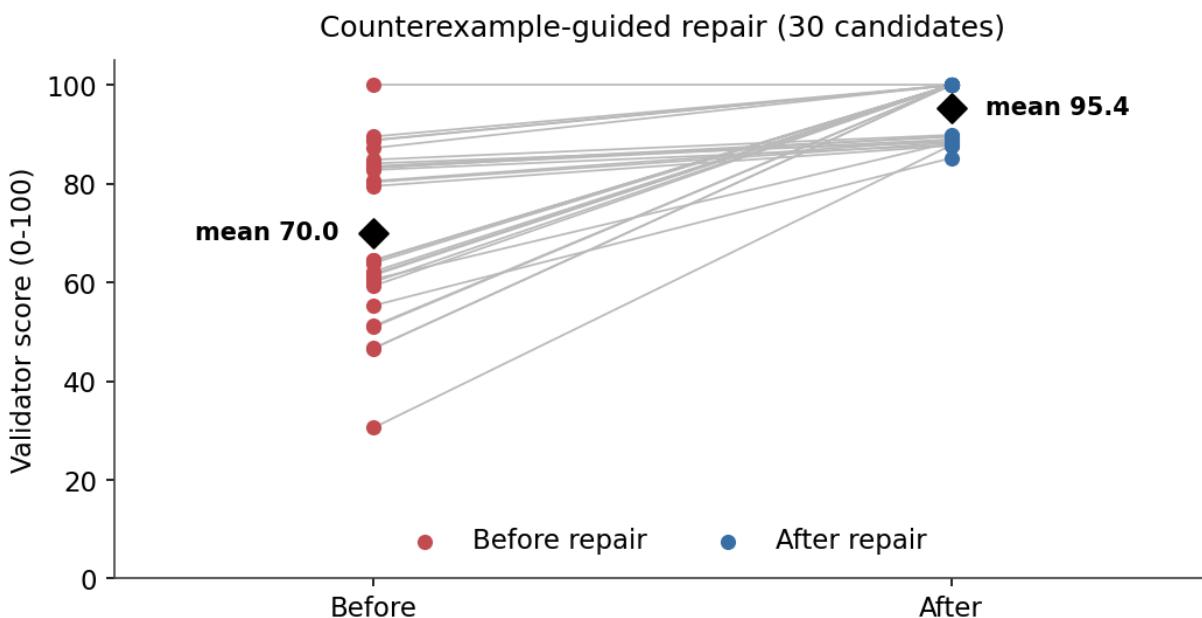


Figure 2. Validator score before and after counterexample-guided repair. Each grey line is one of 30 candidates; diamonds mark the means. Repair lifts the mean from 70.1 to 95.4 and clears every high-severity under-constraint, with one regression.

Live Gemini results: of 30 attempted generations, 20 parsed successfully (66.7 percent), with a mean score of 87.3 and a median of 100. All 10 failures were free-tier rate-limit errors from the API, not malformed model output, so they affected requirements R08 to R10 and reduced coverage of the requirement set rather than reflecting model quality. Across prompt modes, obligation-guided prompting reached full obligation coverage (1.00) against 0.94 for direct translation and 0.97 for adversarial review (Figure 3). Behavioral agreement was similar across modes, so the obligation-guided gain is specifically in not dropping named guards, which is exactly the high-severity failure we care about. The validator also surfaced a concrete unsafe case, a specification that allowed a researcher to access a non-anonymized, sensitive, high-risk record that

the gold policy denied, and returned an SMT counterexample for it.

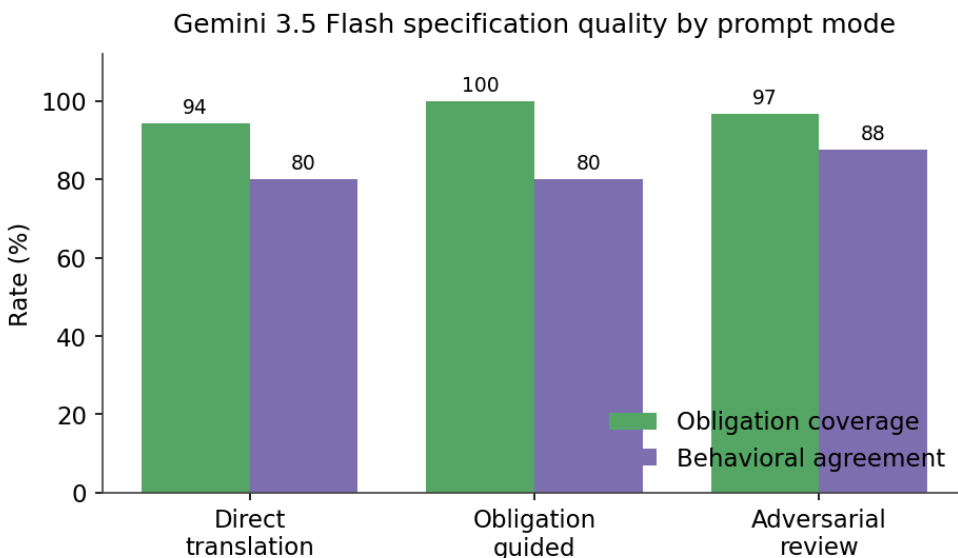


Figure 3. Gemini 3.5 Flash specification quality by prompt mode, over the 20 parsed specs. Obligation-guided prompting reaches full obligation coverage (100 percent) versus 94 and 97 percent, while behavioral agreement stays near 80 percent across modes.

Three points support the claims on robustness. First, results come from 90 controlled objects (30 candidates and 60 mutants) with known injected faults, so detection is measured against ground truth rather than judgment. Second, behavioural testing uses 2,000 inputs per requirement with a fixed seed, and the three independent systematic strategies agree, so the detection numbers are not an artifact of one sampling choice. Third, the repair result is a paired before-and-after on the same candidates, which is why the per-item lines in Figure 2 are the right way to read it. The main fragility is the live LLM portion, where the parse rate and the small parsed set make the prompt-mode comparison suggestive rather than conclusive.

5. Discussion and Limitations

The practical message is that hand-written tests give false confidence for machine-generated safety specifications. A small boundary suite, the kind a developer would plausibly write, missed nearly half of the high-severity faults in our setup, while cheap systematic validation solves the missing link. For any workflow where a language model drafts an access policy, sampling against a reference is a low-cost safeguard that should be the default rather than an afterthought.

The structural message concerns what to do once sampling has already caught most behavioural errors. All three systematic strategies converged on the same detection rate, which tells us that

the formal oracles add little raw detection. What they add is explanation. A reviewer needs the missing guard named and a concrete input that breaks the policy, because that is the information that makes a failure fixable, and our repair result shows that this same information closes the loop back to a corrected specification. As more safety policies are drafted with model assistance, the bottleneck moves from writing specifications toward trusting them, and an explain-and-repair loop is what supports that trust.

Limitations

The setting is a small, finite, single-domain DSL rather than a production policy language, so the absolute detection numbers should be read as evidence within this testbed and not as a general guarantee. The repair stage is a controlled simulation that adds obligations back and substitutes gold logic for recognized over-constraints, so the 96.7 percent success rate shows that counterexamples carry a usable signal and does not show that an arbitrary model would reach that rate on its own. The live evaluation was limited by free-tier rate limits, which dropped a third of generations and left seven requirements with parsed output, so the prompt-mode comparison is suggestive and needs a larger paid run. We also did not address an adversarial author, so a model that crafts a specification to pass the oracles while hiding unsafe behavior falls outside our threat model.

Three assumptions shape how the results should be read. First, we assume the gold specifications are correct, since every oracle compares against them. If a gold specification were itself wrong, that error would pass silently through coverage, agreement, and SMT checks alike, and our detection and repair numbers would overstate true faithfulness. Second, we assume the model is cooperative, and its errors are accidental. If the author were adversarial, the high detection rate would no longer hold, because the oracles check for plausible mistakes rather than concealed ones. Third, we assume the 30 candidates and 60 mutants represent the kinds of errors real models make. If real generation errors fall outside these patterns, the 98.9 percent detection rate would be optimistic, and the true value of the method would depend on how well the oracle set generalizes beyond the faults we designed for.

Future Work

The next steps are a larger paid run across more models and requirements to test whether the obligation-guided coverage advantage holds, a real model-in-the-loop repair where the generator fixes its own specification from the counterexample [13], an adversarial setting where a generator is rewarded for passing the oracles, and a port of the DSL to a standard policy language such as Cedar or XACML so the method can be evaluated against real misconfigurations instead of injected ones [15].

6. Conclusion

We built and evaluated a pipeline for checking whether LLM-generated safety specifications faithfully express their natural-language requirements, using biosecurity access control as a testbed. Across 90 controlled objects, a small manual test suite detected only 63.3 percent of faults and 55.3 percent of high-severity faults, while systematic validation reached 98.9 percent and 100 percent. Counterexample-guided repair fixed 29 of 30 candidates and removed every high-severity under-constraint, and a live run on Gemini 3.5 Flash showed that obligation-guided prompting produced full obligation coverage.

The broader takeaway is that systematic sampling is enough to catch most behavioural errors, while the value of combining obligation coverage, metamorphic relations, and SMT search lies in explanation. The pipeline names the missing guard, returns a concrete counterexample, and turns that counterexample into a repair. As safety policies are increasingly drafted with model assistance, that explain-and-repair loop is what lets a human trust the result.

Code and Data

Code repository: https://github.com/Isabelle9999/sps_spec_validation

The repository contains the full Jupyter notebook implementing the entire pipeline (input space, restricted DSL, the 10 hand-authored requirements with gold specifications and safety obligations, the 30 LLM-style candidates, the 60 controlled mutants, the four oracles, the scoring and repair stages, and the live Gemini evaluation), together with all generated result files: the method comparison, repair results, review cards, prompt-mode summaries, case studies, and the raw Gemini records and parse-failure logs.

References

- [1] Zuo, X., et al. 2025. PAT-Agent: Autoformalization for Model Checking. arXiv preprint. <https://arxiv.org/abs/2509.23675>
- [2] Liu, S., et al. 2025. Evaluating Autoformalization Robustness via Semantically Similar Paraphrasing. arXiv preprint. <https://arxiv.org/abs/2511.12784>
- [3] Tantawy, A., and Abdelgawad, H. 2023. Model Checking Access Control Policies: A Case Study using Google Cloud IAM. arXiv preprint. <https://arxiv.org/abs/2303.16688>
- [4] Bouchet, M., Cook, B., Cutler, B., Druzkina, A., Gacek, A., Hadarean, L., Jhala, R., Marshall, B., Peebles, D., Rungta, N., Schlesinger, C., Stephens, C., Varming, C., and Warfield, A. 2020.

Block Public Access: Trust Safety Verification of Access Control Policies. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20). ACM. <https://doi.org/10.1145/3368089.3409728>

[5] Sissodiya, A., et al. 2025. Formal Verification for Preventing Misconfigured Access Policies. Luleå University of Technology. <https://ltu.diva-portal.org/smash/get/diva2:1991066/FULLTEXT01.pdf>

[6] Author(s) TBD. 2025. AutoICE: Automatically Synthesizing Verifiable C Code via LLM-driven Evolution. arXiv preprint. <https://arxiv.org/abs/2512.07501>

[7] Author(s) TBD. 2026. CktFormalizer: Autoformalization of Natural Language into Circuit Representations. arXiv preprint. <https://arxiv.org/abs/2605.07782>

[8] Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., and Varming, C. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9. IEEE. <https://doi.org/10.23919/FMCAD.2018.8602994>

[9] Hu, V. C., Kuhn, D. R., and Yaga, D. 2017. Verification and Test Methods for Access Control Policies/Models. NIST Special Publication 800-192. National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-192>

[10] Chen, T. Y., Cheung, S. C., and Yiu, S. M. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology. <https://arxiv.org/abs/2002.12543>

[11] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. 2006. Combinatorial Sketching for Finite Programs. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII). ACM. <https://doi.org/10.1145/1168857.1168907>

[12] Jha, S., and Seshia, S. A. 2014. Are There Good Mistakes? A Theoretical Analysis of CEGIS. In Proceedings of the 3rd Workshop on Synthesis (SYNT). <https://arxiv.org/abs/1407.5397>

[13] Orvalho, P., Janota, M., and Manquinho, V. 2025. Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. In Proceedings of the AAAI Conference on Artificial Intelligence. <https://pmorvalho.github.io/uploads/papers/aaai25-LLM-CEGIS-Repair.pdf>

- [14] Jia, Y., and Harman, M. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5), pp. 649–678.
<https://doi.org/10.1109/TSE.2010.62>
- [15] Cutler, J., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Joshi, E., Lam, P., McCutchen, M., et al. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1).
<https://doi.org/10.1145/3649835>
- [16] Author(s) TBD. 2026. Faithful Autoformalization via Roundtrip Verification and Repair. arXiv preprint. <https://arxiv.org/abs/2604.25031>
- [17] Ayon, R. S., and Ahmed, S. 2026. AutoReSpec: A Framework for Generating Specification using Large Language Models. In *Proceedings of the 3rd IEEE/ACM International Conference on AI Foundation Models and Software Engineering (FORGE '26)*. ACM.
<https://doi.org/10.1145/3793655.3793731>
- [18] Author(s) TBD. 2026. Towards an Agentic LLM-based Approach to Requirement Formalization from Unstructured Specifications. arXiv preprint.
<https://arxiv.org/abs/2604.18228>

Appendix

A. Prompts used for the three modes

All three modes share one base prompt. Only the final "mode instruction" line differs, which isolates the effect of prompting strategy. The shared prompt supplies the requirement, the available fields, the DSL operators, the required JSON schema, a format example, and a fixed set of rules. The model used was Gemini 3.5 Flash through the Gemini API, temperature 0.7, with structured JSON output requested first and a plain-JSON fallback.

The mode-specific instruction lines are:

Direct translation: "Translate the natural-language requirement directly into one restricted DSL specification."

Obligation-guided: "Use the listed obligations as traceability anchors and generate a DSL specification that covers all obligations." In this mode the prompt also lists each obligation with its id, name, severity, and predicate.

Adversarial review: "Act as a strict safety reviewer and generate a conservative specification, but do not add unrelated restrictions such as PI-only access unless the requirement says so."

The base prompt also fixed several rules that applied in every mode: output JSON only, do not output Python, do not introduce fields outside the available list, use only the operators AND, OR, NOT, EQ, NEQ, GE, GT, LE, LT, FIELD, and CONST, prefer a safer deny-by-default reading under ambiguity without adding unrelated PI-only restrictions, and return exactly one candidate.

B. The five metamorphic relations

Each relation perturbs one safety-relevant dimension of an input and defines the response that would violate a conservative access-control intuition. A violation means the specification responded unsafely to that controlled change.

1. Clearance monotonicity. Raising a user's clearance by one level should never turn an allow into a deny. Violation if the base input is allowed and the higher-clearance follow-up is denied.
2. Risk monotonicity. Raising a record's risk level by one should never turn a deny into an allow. Violation if the base is denied and the higher-risk follow-up is allowed.
3. Expiration safety. Marking a user's approval as expired should never turn a deny into an allow. Violation if the base is denied and the expired follow-up is allowed.
4. External anonymization relaxation. For an external user, anonymizing a record should never turn an allow into a deny. Violation if the base is allowed and the anonymized follow-up is denied.
5. Emergency logging safety. During an emergency, removing logging should never turn a deny into an allow. Violation if the base is denied and the unlogged follow-up is allowed.

C. Worked review card

The following is the validator's full computation for candidate R10-C3, an over-permissive Boolean connective fault, which illustrates how the four oracles combine into the final score.

Coverage rate 0.0, with all five obligations missing and two of them high severity. Behavioral agreement 0.4235 over 2,000 sampled inputs, with 1,153 disagreements, of which 1,134 are under-constrained and 799 of those are high-severity under-constrained. Metamorphic consistency 1.0, no relation violations. SMT consistency 0.0, since the solver found a disagreeing input. The score is then $25(0.0) + 25(0.4235) + 20(1.0) + 20(0.0) + 10(0.0) = 30.59$, and the candidate is labeled high risk because it both misses high-severity obligations and allows unsafe access. After repair this candidate rose to 87.84.

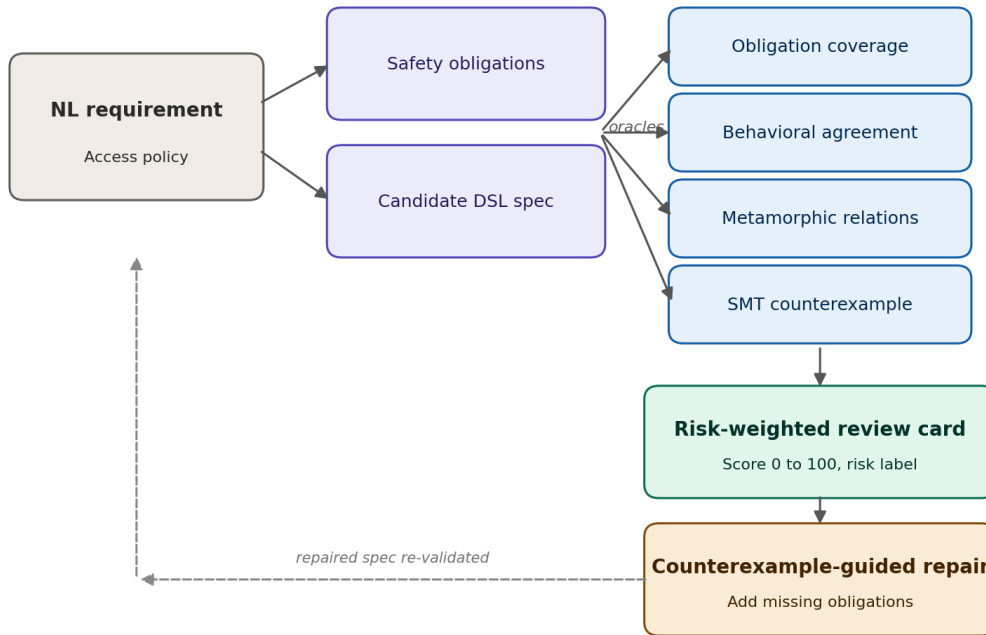
D. Case studies

Case 1, under-constrained, the most safety-critical pattern. Requirement: external collaborators may access only anonymized, non-sensitive, low-to-moderate-risk records through logged access. On the input of a researcher with clearance 4 viewing a non-anonymized, sensitive, risk-level-4 record with logging on, the gold specification denies access while the candidate allows it. The candidate has lost a guard that blocks unsafe access.

Case 2, metamorphic violation. Requirement: high-risk biological records are accessible only to PIs with sufficient clearance, completed training, and logged access. Under the risk-monotonicity relation, raising a record from risk level 3 to risk level 4 for the same external user flipped the candidate's decision from deny to allow, which a safe policy should never do. The candidate missed the PI-only obligation, and the relation exposed the inconsistency even though sampled behavioral agreement looked high.

A real Gemini case also surfaced. The validator found a specification that allowed a researcher to access a non-anonymized, sensitive, high-risk record that the gold policy denied, and returned an SMT counterexample witnessing it.

E. System diagram and reproducibility



The pipeline figure above shows the full flow from a natural-language requirement to obligations and a candidate DSL specification, through the four oracles into the risk-weighted review card, and then into the counterexample-guided repair loop that re-validates the repaired specification. The complete experiment, including the gold specifications, obligations, mutant generation, oracle

implementations, scoring, repair, and the live Gemini stage, runs from a single notebook with a fixed random seed. The full review cards, mutant set, raw Gemini records, and parse-failure logs are in the repository.

LLM Usage Statement

The Gemini API was a core experimental component, generating the specifications evaluated in the live experiment. Figures were generated by Claude. For writing the report, LLM assistance was used for editing. The validation pipeline was built independently.