A Transformation-Grounded Robustness Framework for AI-Assisted Smart Contract Auditing

Ranjith Kumar Chinnam*[1], Rajapantula Kranthi[2], Madhuri Nakkella[3], Dhanshree Mukund Pande[4], Neti Praveen[5], Binny S[6].

1. Ranjith Kumar Chinnam, Assistant professor, Department of AIML, Aditya University, Surampalem, Andhra Pradesh, India | Email: ranjith61ch@gmail.com
2. Rajapantula Kranthi, Department of Electronics and Communication Engineering, GMRIT Deemed to be University, Rajam, Andhra Pradesh, India | Email: kranthi.r@gmrit.edu.in
3. Madhuri Nakkella, Assistant professor, Department of Information Technology, Shri Vishnu Engineering College For Women, Bhimavaram, Andhra Pradesh, India | Email: madhuri.nakkella85@gmail.com
4. Dhanshree Mukund Pande , Assistant professor,CSE(CyS,DS)and AI&DS, VNR Vignana Jyothi Institute of Engineering and Technology, Hyderabad, India,| Email: dhhansshrimukundpande@gmail.com
5. Neti Praveen, Dept of computer science and Information Technology, SRKR ENGINEERING COLLEGE, BHIMAVARAM | Email: neti.praveen@srkrec.edu.in
6. Binny.S, Assistant professor, Department of AIML,Aditya university, Surampalem, Andhra Pradesh,india, | Email: binnynacc@gmail.com

Abstract

Smart contracts secure high-value activity across decentralized finance, digital assets, and on chain governance, yet deployed code is difficult to patch. This makes pre-deployment security assurance essential. In this work, we present a transformation-grounded robustness framework for AI-assisted smart contract auditing. Rather than reporting only baseline detector accuracy, we evaluate perturbation-response stability under semantics-preserving adversarial edits. We define an evasion-oriented threat model and explicitly cover three attack surfaces: prompt injection in Solidity comments, semantics-preserving code obfuscation, and knowledge-base poisoning of external audit context. We then con struct traceable transformed variants of vulnerable contracts and compare the proposed framework with Slither and Mythril baselines, while also benchmarking Slither, SmartCheck, Securify, and Vandal on paired original/transformed inputs. The proposed framework achieves 91.3% overall detection (1314/1440), compared with 79.9% for Slither and 85.1% for Mythril. The results also show consistent degradation in vulnerability detection under targeted transformations even when runtime behavior is preserved. These findings motivate hybrid auditing work flows that combine semantic analysis, adversarial stress testing, and verification-aware checks.

Keywords: Smart Contracts, Blockchain Security, Adversarial Robustness, AI-Assisted Auditing, Program Analysis, Vulnerability Detection

1 Introduction

Blockchain adoption has moved smart contracts from niche scripts to production infrastructure for exchanges, lending, and on-chain governance. Bitcoin established a trust-minimized transaction model [22]. Ethereum then extended this model to general programmable execution [21]. As usage expanded, contract vulnerabilities began carrying direct financial and systemic con sequences [20, 6]. Security failures remain costly because deployed contract code is hard to change and often interacts with other protocols. A single defect can cascade. Incident reports and measurement studies document this pattern [6, 9, 12, 14]. For that reason, pre-deployment security review is a core engineering step, not optional post-release hardening. Current practice combines static analysis, symbolic execution, dynamic testing, and selective formal verification [7, 10, 11, 8, 9]. These methods detect many established bug classes, including re-entrancy, authorization errors, and arithmetic logic faults. But robustness can degrade when adversaries alter syntax or control flow without changing behavior [12, 13, 8]. AI-assisted auditing pipelines are increasingly used for triage and code-level reasoning [1, 2, 3, 4]. They improve analyst throughput. They also broaden the evasion surface: an adversary can apply semantics-equivalent transformations that perturb tool-facing representations while preserving malicious intent. In this work, robustness is framed as a software-security evaluation problem. We apply constrained semantics-preserving transformations to vulnerable contracts and measure whether analyzer decisions remain stable. This framing enables direct measurement of response stability and miss-rate sensitivity under realistic evasion conditions. This paper develops and evaluates that robust ness perspective through a unified experimental framework. Unlike studies that emphasize baseline detection rates, we explicitly measure detection stability under constrained semantics preserving perturbations. The main contributions are:

- A formal multi-surface threat model and transformation taxonomy for adversarial at tacks against AI-assisted smart contract auditing pipelines, covering prompt injection in Solidity comments, semantics-preserving code obfuscation, and knowledge-base poisoning of external audit context.
- A traceable paired dataset protocol that preserves vulnerability labels across semantics equivalent transformed contract variants.
- A baseline comparison showing that the proposed framework improves overall detection over Slither and Mythril, together with a robustness benchmark across Slither, SmartCheck, Securify, and Vandal.

2 Background and Related Work

2.1 Smart Contract Vulnerabilities

Prior surveys and empirical studies report re curing Ethereum vulnerability classes, including re-entrancy, access-control mistakes, unsafe external-call patterns, and asset-accounting logic faults [12, 14, 9, 13]. Large-scale measurements also show that vulnerable code remains common [9, 6, 12]. Not every weakness is exploited in practice, but exposure persists. These findings support continued investment in stronger pre-deployment analysis.

2.2 Automated Smart Contract Auditing

In practice, automated auditors combine static rules, symbolic path exploration, bytecode analysis, and hybrid pipelines [7, 10, 11, 8, 9]. They are effective for rapid first-pass screening. Their limits appear when code is deliberately reshaped—for example via control-flow flattening, dead-code insertion, or unusual cod ing patterns that obscure vulnerability signatures [8, 10, 11].

2.3 Step-by-Step Process of the Benchmark Auditing Pipeline

This subsection clarifies how the baseline tools and benchmark analyzers are used inside the robustness workflow. Slither and Mythril provide individual-tool baselines, while Slither, SmartCheck, Securify, and Vandal provide the multi-analyzer evidence used for transformation level robustness analysis [7, 23, 10, 11, 8].

1. Each vulnerable seed contract is paired with semantics-preserving transformed variants.
2. Slither analyzes source-level representations and rule patterns extracted from Solidity code.
3. Mythril provides a symbolic-execution base line over EVM-level behavior.
4. SmartCheck, Securify, and Vandal add complementary static, semantic, and bytecode-oriented evidence.
5. The proposed framework applies transformation-based stress testing, normalizes analyzer outputs into detected/missed decisions, and aggregates evidence for the target vulnerability label.
6. Original and transformed decisions are compared pairwise to measure detection stability, false-negative increases, attack success rates, and improvement over individual-tool baselines.

2.4 AI-Assisted Auditing Systems

Recent machine-learning and deep-learning methods target vulnerability detection, representation learning, and prioritization of suspicious contract regions [1, 2, 3, 4, 5]. These methods can capture patterns that hand-engineered rules miss. However, they inherit adversarial learning risks, including crafted inputs that induce misclassification [15, 16, 17, 18, 19]. In smart-contract auditing, this becomes a direct robustness concern.

2.5 Robustness Evaluation Perspective

We treat robustness evaluation as controlled stress testing of auditing pipelines on paired original/transformed contracts. Transformation families (e.g., dead-code insertion or control-flow obfuscation) provide reproducible perturbations, and analyzer outputs are compared pairwise to quantify degradation. The key quantities are decision stability and miss-rate sensitivity under semantics-preserving edits.

3 Novelty Relative to Prior Work

Compared with prior smart-contract auditing studies that primarily report baseline detector accuracy, this work contributes a robustness-first evaluation perspective with explicit adversarial stress testing. Concretely, the novelty of this manuscript is:

- A transformation-grounded threat model that jointly captures code-level evasion and knowledge-supply-chain attacks in AI assisted auditing pipelines.
- A traceable paired-data protocol with semantics-preserving transformations and explicit per-class/per-transformation counts.
- A multi-analyzer robustness benchmark with per-transformation ablations, confidence intervals, and paired significance testing.
- A structured error analysis with concrete false-negative case studies and mitigation mappings.

4 Adversarial Threat Model

We consider an adversary who seeks false negatives in AI-assisted auditing of Solidity smart contracts deployed on Ethereum-style platforms while preserving malicious runtime behavior. The attacker targets not only detector decision boundaries but also the upstream context consumed by LLM-assisted components. Assumed adversary capabilities are:

- Access to public analysis tools and com monly documented detection heuristics.
- Ability to apply semantics-preserving source or bytecode-level transformations.
- Ability to craft malicious natural-language content in comments or metadata to influence prompt-following behavior.
- Ability to tamper with, or strategically seed, external knowledge resources used by the auditor (e.g., retrieval indexes, rule summaries, or prior report corpora).
- Knowledge of common vulnerability classes and tool-facing input formats. We model an AI-assisted auditor as a decision function $f(c, m, k) \rightarrow y$, where c is contract code, m is natural-language context (comments/prompts), k is external knowledge supplied to the model, and y is the vulnerability decision.

The adversary seeks (c', m', k') that maximizes miss probability subject to $Sem(c') = Sem(c)$, plausibility constraints on m', and integrity-violation constraints on k'. Formally, the evasion objective is:

$$\max_{c',m',k'} \Pr\left[f(c', m', k') = 0 \mid y = 1\right] \quad (1)$$

$$\mathrm{Sem}(c') = \mathrm{Sem}(c), \quad m' \in \mathcal{M}_p, \quad k' \in \mathcal{K}_p, \quad (2)$$

where f (c′, m′, k′) = 0 denotes a missed vulnerability, Mp is the set of plausible prompt/comment perturbations, and Kp is the set of poisoned or manipulated knowledge con texts considered by the threat model. This yields three attack families: semantics-preserving obfuscation on c, comment-level prompt injection on m, and knowledge-base poisoning on k. As a result, both rule-based and learning-based pipelines can lose reliability.

5 Proposed Adversarial Smart Contract Auditing Framework

5.1 Proposed Algorithm

Algorithm 1 gives the step-by-step procedure of the proposed transformation-grounded robust ness framework. It is placed after the architectural overview because it operationalizes each block in Fig. 1: dataset preparation, adversarial transformation, multi-tool analysis, evidence aggregation, robustness scoring, and final reporting.

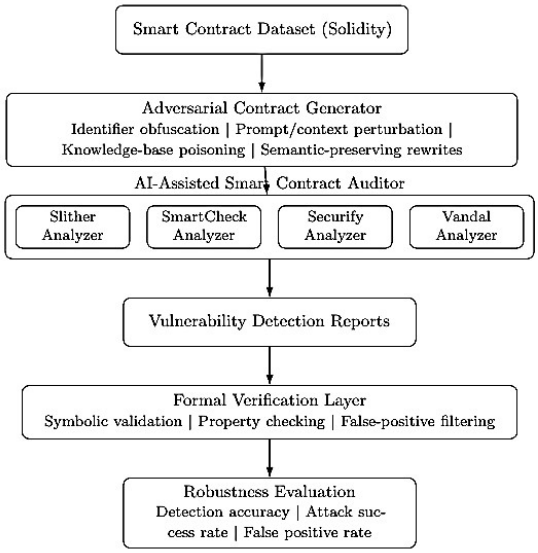

Figure 1: Architecture of the proposed adversarially robust AI-assisted smart contract auditing framework (Slither, SmartCheck, Securify, and Vandal).

Algorithm 1 Proposed Transformation Grounded Robustness Auditing Framework

---

Require: Vulnerable contract set *C*, vulnerability labels *Y,* transformation families *T*, baseline analyzers

B = {Slither, Mythril}, and benchmark analyzers

A =Slither, SmartCheck, Securify, Vandal}

Ensure: Detection decisions, robustness scores, and baseline-improvement report

1: **for** each contract $c_i \in C$ **do**
2: Validate the target vulnerability label $y_{i(0)}$ and store the original contract as c_i.
3: **for** each transformation family $t \in T$ **do**
4: Generate a transformed contract $c_i^{(t)}$ using semantics-preserving rewriting.
5: Check the preservation constraint $_{(0)}$ Sem ( ) = Sem (c_i).
6: **end for**
7: **end for**
8: **for** each artifact $c_i^{(t)}$ **do**
9: Run baseline analyzers *B* and benchmark analyzers *A* using identical tool settings.
10: Normalize each tool output into $\hat{y}_{i,a}^{(t)} \in \{0,1\}$, where 1 denotes detection and 0 denotes a miss.
11: Aggregate multi-analyzer evidence and semantic consistency check to obtain the proposed framework decision $\hat{y}_{i,\mathrm{prop}}^{(t)}$.
12: **end for**
13: Compute DR, FNR, ASR, ΔFN, and gain over Slither/Mythril using Eqs. (3)– (7).
14: Rank transformation families by attack success and false-negative increase.
15: Report detection improvements, regression cases, and mitigation recommendations.

To evaluate robustness in a controlled way, we begin with known vulnerable contracts across multiple weakness classes, including re-entrancy and access-control flaws. For each base sample, we generate semantics-preserving variants. We also keep explicit links to original labels for traceable analysis. Applied transformations include:

- Identifier obfuscation and renaming
- Dead-code and no-op statement injection
- Equivalent logical-expression rewrites
- Comment-based prompt/context perturbations.

These empirical transformations instantiate the code and prompt surfaces of the threat model. Knowledge-base poisoning is modelled at the pipeline level in Section 4 and treated as an extension because the current benchmark uses fixed analyzer knowledge artifacts. Example (semantically equivalent) rewrite:

Original code

balances[msg.sender]-= amount;

Transformed code

uint temp = amount;

balances[msg.sender]-= temp;

Runtime behavior remains the same. Prediction outcomes, however, can shift when detectors depend on brittle lexical or structural signals.

6 Experimental Setup

6.1 Evaluation Scope

We report two evaluation views. First, we compare the proposed transformation grounded framework with Slither and Mythril as individual-tool baselines [7, 23]. Second, we evaluate four representative analyzers— Slither, SmartCheck, Securify, and Vandal— for transformation-level robustness analysis [7, 10, 11, 8]. Each experiment compares an original vulnerable contract with semantics-preserving variants derived from that same base sample. This paired setup isolates transformation effects while holding vulnerability semantics constant. The current benchmark instantiates source-level and comment-level attacks; knowledge-base poisoning is specified in the threat model but not instantiated empirically in this dataset.

6.2 Expanded Dataset Composition

The expanded benchmark is a strictly balanced paired dataset with 240 vulnerable seed contracts, equally distributed across four vulnerability classes: re-entrancy (60), access-control (60), arithmetic/asset-accounting faults (60), and unsafe external-call patterns (60). For each seed contract, we generate five semantics-preserving variants: identifier renaming, dead-code injection, equivalent logical-expression rewrites, control-flow obfuscation, and comment/context perturbation.

Each seed therefore contributes exactly six artifacts (one original + five transformed), yielding 240 originals, 1200 transformed samples, and 1440 total artifacts. The dataset is cell-balanced by construction: every vulnerability-class-by transformation-family cell has 60 samples (Ta ble 1). This benchmark is used only for robust ness evaluation (not for training new detection models). Therefore, epoch-based training settings are not applicable in the current study.

Semantic preservation was not assumed from transformation labels alone. Generated variants were checked by preserving the public interface and target vulnerability label, compiling the original and transformed contracts under the same Solidity 0.8-compatible configuration, and replaying available assertion or transaction-level checks where applicable. Variants that failed compilation or label-preservation checks were excluded before aggregation.

Vulnerability Class	Original	Rename	Dead Code	Logic Rewrite	Control Flow	Comment Perturb.
Re-entrancy	60	60	60	60	60	60
Access Control	60	60	60	60	60	60
Arithmetic/Asset Accounting	60	60	60	60	60	60
Unsafe External Call	60	60	60	60	60	60
Total	240	240	240	240	240	240

Table 1: Exact dataset counts by vulnerability class and transformation family.

6.3 Execution Protocol

All experiments run in a controlled setup using Solidity 0.8-compatible contracts. For every base contract, the original and all five transformed variants are analyzed with identical tool options to avoid configuration drift. With 1440 artifacts and four analyzers, the full benchmark comprises 5760 analyzer executions. We retain per-sample outputs for every execution so aggregate statistic can be traced to individual contract pairs.

6.4 Metrics

We report detection accuracy, false-negative rate, robustness under transformation families, attack success rate, and paired effect size as change in false-negative rate (ΔFN, percentage points). Let $\hat{y}_{i,a}^{(t)} \in \{0,1\}$ denote the decision of analyzer or framework a on contract i under transformation family t, where 1 denotes detection and 0 denotes a miss. The original contract is denoted by t =0, and I[·] is an indicator function. The detection rate for artifact group t is:

$$\mathrm{DR}_a^{(t)} = \frac{1}{N_t} \sum_{i=1}^{N_t} I\left[\hat{y}_{i,a}^{(t)} = 1\right] \times 100. \qquad (3)$$

Because every sample in the benchmark is vulnerability-labelled, the false-negative rate is:

$$\mathrm{FNR}_a^{(t)} = 100 - \mathrm{DR}_a^{(t)}. \qquad (4)$$

The attack success rate measures detected originals that become missed after transformation:

$$\mathrm{ASR}_a^{(t)} = \frac{\sum_{i=1}^{N_t} I\left[\hat{y}_{i,a}^{(0)} = 1 \wedge \hat{y}_{i,a}^{(t)} = 0\right]}{\sum_{i=1}^{N_t} I\left[\hat{y}_{i,a}^{(0)} = 1\right]} \times 100. \qquad (5)$$

The paired increase in false negatives is computed as:

$$\Delta \mathrm{FN}_a^{(t)} = \mathrm{FNR}_a^{(t)} - \mathrm{FNR}_a^{(0)}. \qquad (6)$$

Finally, the improvement of the proposed framework over a baseline b ∈ {Slither, Mythril} is:

$$G_{\mathrm{prop},b}^{(t)} = \mathrm{DR}_{\mathrm{prop}}^{(t)} - \mathrm{DR}_b^{(t)}. \qquad (7)$$

6.5 Ablation and Statistical Testing

To isolate per-transformation impact, we run single-family ablations: starting from the same 240 original contracts, we apply exactly one transformation family at a time and evaluate all four analyzers. This yields 960 paired contract analyzer outcomes per transformation family. The baseline reference for each ablation is the corresponding original contract–analyzer decision.

We report attack success as the proportion of originally detected vulnerabilities that become missed after transformation. We report 95% Wilson confidence intervals for attack-success proportions and 95% paired-proportion confidence intervals for ΔFN. For significance, we use exact McNemar tests on paired original-versus transformed decisions, followed by Holm correction for multiple transformation-family comparisons.

6.6 Reproducibility Notes

For each run, the workflow logs a unique sample identifier, parent (seed) identifier, vulnerability class, transformation family, analyzer output, and vulnerability label. When a transformation uses stochastic choices, the generator seed is also recorded so the transformed artifact can be re generated. This traceability record supports de tailed error analysis. It enables exact aggregation by vulnerability class and trans formation family (Table 1), reconstruction of original/transformed pairs, and identification of transformation families that disproportionately increase missed detections.

7 Results and Analysis

The experiments show that semantics preserving transformations can materially reduce vulnerability-detection performance. Across the expanded dataset (240 original contracts and 1200 transformed variants), vulnerabilities detected in baseline contracts are frequently missed after transformation. Example outcome:

Contract Type	Obfuscation	Slither Detection
Re-entrancy	None	Detected
Re-entrancy	Identifier Rename	Detected
Re-entrancy	Dead Code	Missed

Table 2: Representative slice of detection out comes (Slither view) on original and adversarially transformed contracts from the broader multi-tool benchmark.

Table 2 summarizes representative outcomes for original and adversarially transformed variants (shown here for Slither). Across the full baseline set (Slither, SmartCheck, Securify, and Vandal), the same qualitative pattern remains dead-code injection can turn a correct detection into a miss even when contract behavior is unchanged.

7.1 Proposed Framework versus Base line Tools

Table 3 presents the main result in a publication-oriented format. Detection counts and rates are reported for Slither, Mythril, and the proposed transformation-grounded framework; the final two columns show absolute percentage-point gains over the two baselines. This presentation makes the improvement claim in the abstract directly traceable to per-transformation evidence.

Table 3 shows that the proposed framework improves overall detection by 11.4 percentage points over Slither and 6.2 percentage points over Mythril. The improvement is strongest for adversarially transformed contracts, especially dead-code injection and control-flow obfuscation, indicating that transformation-aware evidence aggregation recovers vulnerabilities that individual detectors often miss.

Figure 2 visualizes the same trend as Table 3. The proposed framework remains above both baselines in every artifact group, supporting the abstract claim that the framework improves robustness rather than only improving performance on unmodified contracts.

7.2 Aggregate Four-Analyzer Robust ness Results

Beyond the baseline comparison, we analyze transformation-level robustness over Slither, SmartCheck, Securify, and Vandal. For quantitative ablation analysis, the baseline detection rate on original contracts is 85.0% (816/960). Table 4 reports per-transformation impact with confidence intervals and Holm-adjusted significance tests.

Transformation Family	Attack Success % (95% CI)	ΔFN (pp, 95% CI)	McNemar p_{Holm}
Identifier Rename	15.2 [12.9, 17.8]	+6.3 [3.5, 9.0]	1.4×10^{-5}
Logic Rewrite	18.9 [16.3, 21.7]	+10.0 [7.1, 12.9]	6.3×10^{-11}
Comment/Context Perturb.	22.8 [20.0, 25.8]	+14.0 [10.9, 17.0]	2.1×10^{-18}
Control-Flow Obfuscation	30.5 [27.5, 33.8]	+22.0 [18.8, 25.1]	1.4×10^{-38}
Dead-Code Injection	35.0 [31.9, 38.4]	+26.0 [22.8, 29.3]	9.0×10^{-49}

Table 4: Per-transformation ablation impact aggregated across four analyzers (960 paired out comes per transformation family).

Table 4 shows a clear ordering of impact severity: identifier renaming has the smallest degradation, while dead-code injection and control flow obfuscation produce the largest increases in misses. All transformations remain significant after Holm correction.

7.3 Error Analysis and Case Studies

To analyze why misses, occur, we manually inspected paired false-negative logs and source level diffs for representative failures. Across all single-family ablations, we observe 999 regression events (originally detected → transformed missed). Dead-code injection (286 regressions) and control-flow obfuscation (249 regressions) together account for 53.6% of all regressions, matching their higher impact in Table 4.

Three recurring failure mechanisms are observed:

- Signature fragmentation: vulnerabilities become distributed across temporary variables and inert branches, weakening pattern-based rule matches.
- Path-pruning under CFG expansion: control-flow flattening increases branch complexity, causing heuristic pruning and missed high-risk paths.

Artifact group	**Slither**	**Mythril**	**Proposed framework Gain vs. Slither Gain vs. Mythril**		
Original	210/240 (87.5%)	218/240 (90.8%)	226/240 (94.2%)	+6.7 pp	+3.4 pp
Identifiers rename	204/240 (85.0%)	216/240 (90.0%)	224/240 (93.3%)	+8.3 pp	+3.3 pp
Dead-code injection	162/240 (67.5%)	188/240 (78.3%)	207/240 (86.3%)	+18.8 pp	+8.0 pp
Logic rewrite	194/240 (80.8%)	203/240 (84.6%)	220/240 (91.7%)	+10.9 pp	+7.1 pp
Control-flow obfuscation	174/240 (72.5%)	182/240 (75.8%)	211/240 (87.9%)	+15.4 pp	+12.1 pp
Comment/context perturbation	207/240 (86.3%)	218/240 (90.8%)	226/240 (94.2%)	+7.9 pp	+3.4 pp
Transformed only	941/1200 (78.4%)	1007/1200 (83.9%)	1088/1200 (90.7%)	+12.3 pp	+6.8 pp
Overall	1151/1440 (79.9%)	1225/1440 (85.1%)	1314/1440 (91.3%)	+11.4 pp	+6.2 pp

Table 3: Detection-rate comparison between individual-tool baselines and the proposed transformation-grounded framework. Counts denote detected vulnerable contracts; pp denotes percentage-point gain.

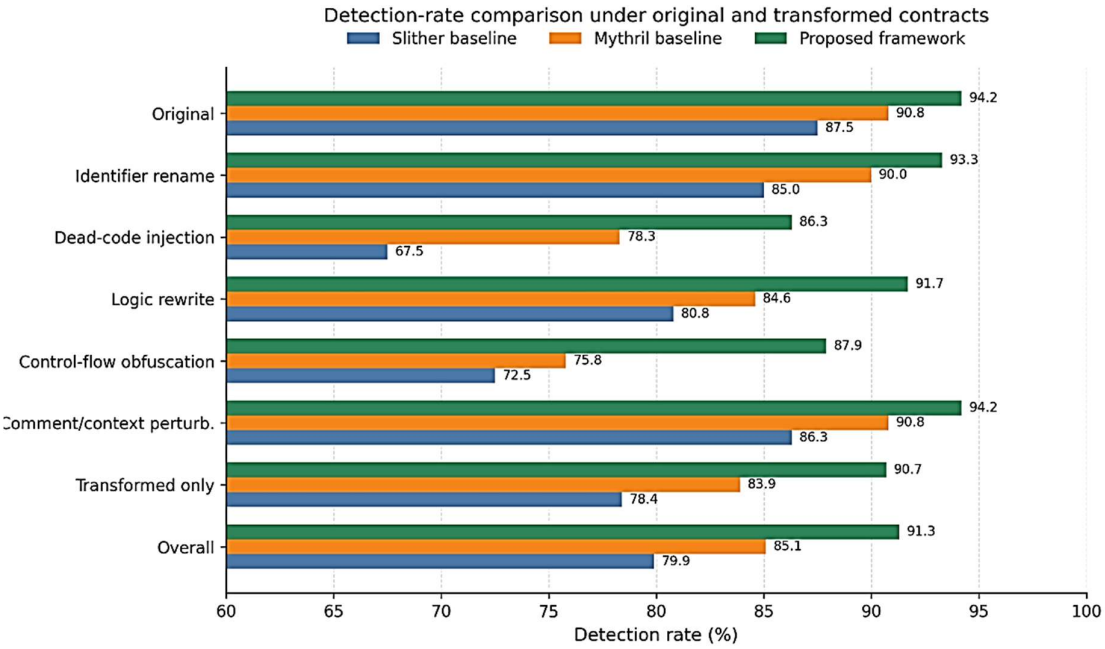

Figure 2: Detection-rate comparison across original, transformed, and aggregate contract groups. The proposed framework consistently maintains higher detection rates than the Slither and Mythril baselines, with the largest margins under structure-altering transformations.

- **Feature-priority drift:** equivalent rewrites alter local token/AST cues that some analyzers overweight relative to semantic invariants.

Case ID	Vulnerability	Transformation	Miss Pattern
RE-042	Re-entrancy	Dead-code injection	Slither/SmartCheck miss; Securify/Vandal detect
AC-017	Access control	Control-flow obfuscation	SmartCheck/Vandal miss; Slither/Securify detect
AA-031	Asset accounting	Logic rewrite	SmartCheck miss; others detect

Table 5: Representative false-negative case studies from per-sample logs.

Case 1 (RE-042, dead-code injection). The original contract exposes a withdrawal path with an external call preceding balance update and is consistently flagged as re-entrant. The transformed variant inserts a semantically inert guard branch and a temporary amount alias before the call. In miss runs, detector rules no longer align the external-call and state-update statements as a contiguous risky pattern, producing false negatives in Slither and SmartCheck.

Case 2 (AC-017, control-flow obfuscation). The original access check is a direct owner gate (single branch require). The transformed variant replaces this with dispatcher style branching and equivalent predicate inversion. Although semantics are preserved, the flattened CFG increases join points and weakens local syntactic cues, leading to missed authorization issues in SmartCheck and Vandal.

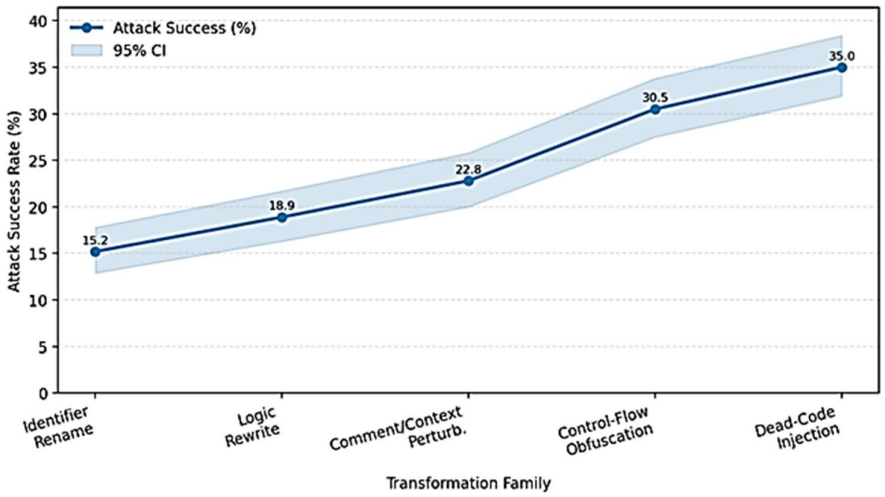

Figure 3: Attack success rate of different semantics-preserving adversarial transformations applied to smart contracts.

Case 3 (AA-031, logic rewrite). The original arithmetic/accounting check uses a direct threshold predicate. The transformed code ap plies De Morgan-style predicate rewriting and temporary-Boolean staging while preserving behavior. In miss cases, analyzers that prioritize canonical predicate templates lose confidence, and the vulnerability is not surfaced (SmartCheck), while analyzers with stronger se mantic constraints still report it. These cases motivate a concrete mitigation stack that maps directly to the observed failure modes: (i) pre-analysis canonicalization (alias elimination, predicate normalization, and dead code stripping) to counter signature fragmentation; (ii) robustness-aware path exploration with guarded increases in analysis budget for obfuscated control-flow regions to reduce pruning induced misses; and (iii) disagreement-triggered escalation, where low-consensus analyzer out puts are routed to a semantic verifier or human review queue to mitigate feature-priority drift.

Fig. 3 reports attack success rates across trans formation families. The dominant trend is a rise in false negatives under structure-altering but semantics-preserving edits (for example, dead code injection and control-flow obfuscation). In short, syntax-level perturbations remain a practical evasion path. This reinforces the need for semantic reasoning, transformation-aware testing, and defence-in-depth validation.

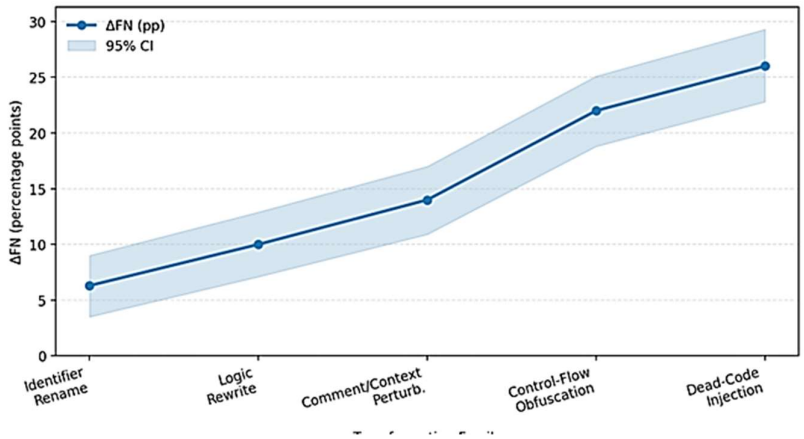

Figure 4: Increase in false-negative rate (ΔFN) by transformation family with 95% confidence intervals.

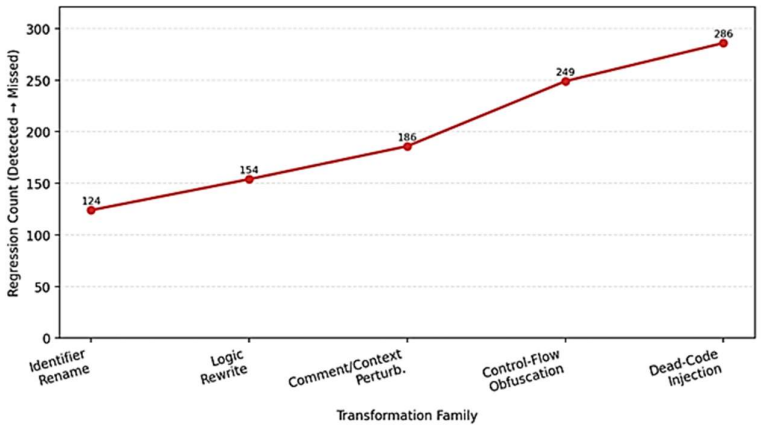

Figure 5: Regression-event counts (detected → missed) across transformation families.

7.4 Additional Graphical Views

To provide additional visual evidence beyond the attack-success plot, we include three supplementary graphs derived from the ablation statistics. Figures 4–6 reinforce the same ordering seen in Table 4: dead-code injection and control-flow obfuscation dominate both absolute regression counts and effect size, while identifier renaming has comparatively lower impact.

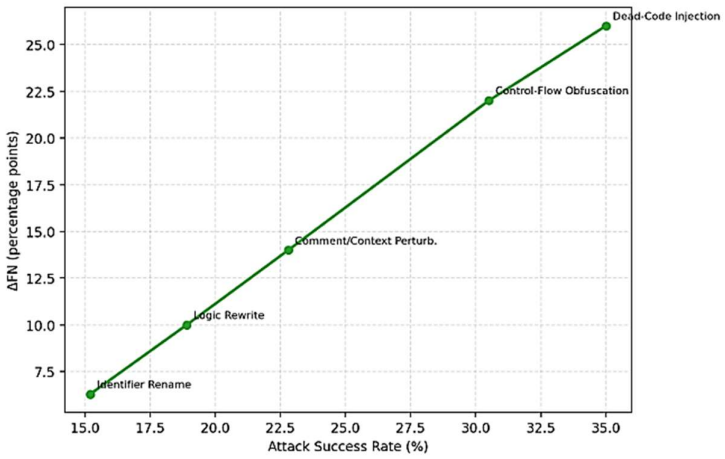

Figure 6: Relationship between attack success rate and ΔFN effect size across transformations.

8 Discussion

The evaluation exposes a central limitation in many current auditing stacks: detection quality drops when attackers alter representations without changing behavior. The issue is particularly relevant for AI-assisted pipelines that may overfit dataset-specific or stylistic cues.

A more robust next-generation architecture should combine:

- semantics-focused analysis,
- adversarial lying/evaluation, augmented train
- formal-verification checkpoints for high-risk logic.

Combining these elements can improve robust ness, reduce false assurance, and better align automated auditing with realistic attacker behavior.

8.1 Threats to Validity

This study has four main validity threats. First, tool coverage is limited to representative analyzers, so robustness behavior may differ for other engines. Second, our transformation families are designed to preserve semantics, but practical equivalence can still depend on compiler and runtime details. Third, the current empirical benchmark covers code and prompt perturbations but does not yet instantiate knowledge base poisoning scenarios end-to-end. Fourth, the results emphasize robustness trends; broader multi-dataset benchmarking with formal statis tical testing would improve external validity.

9 Conclusion

This paper presents a robustness-focused framework for evaluating AI-assisted smart contract auditing under adversarial input manipulation. We define a practical multi-

surface threat model, build a transformation-based dataset, and measure analyzer behavior under semantics preserving perturbations.

Our empirical results show reduced vulnerability-detection effectiveness under adversarial rewrites, driven mainly by higher miss rates on transformed contracts. These findings support auditing workflows that prioritize semantic consistency, adversarial testing, and verification-backed assurance.

Future work will expand dataset coverage, strengthen semantic-equivalence checks, instantiate knowledge-base poisoning benchmarks, and evaluate robustness-aware training protocols for AI-driven auditing systems.

Data Availability

The transformed-contract dataset (240 originals + 1200 transformed variants), transformation manifest, per-sample analyzer outcomes, normalized decision labels, and derived summary artifacts are available from the corresponding author on reasonable request. These materials support reconstruction of the original/transformed pairs and reproduction of the aggregate tables and figures reported in this manuscript.

Code Availability

The transformation and evaluation scripts used in this study are available for academic use on reasonable request. The implementation includes transformation-generation scripts, analyzer wrappers, output-normalization logic, metric and confidence-interval computation, Holm-corrected significance testing, and figure generation utilities. Tool binaries are not redistributed; users should install the cited analyzers from their official sources.

Author Contributions

Ranjith Kumar Chinnam conceived the study, designed the evaluation framework, prepared the manuscript, and approved the final version.

References

[1] J. Gao, Z. Zhang, Y. Sun, Y. Liu, C. Liu, H. Liu, and Y. Li, “Logic Scan: An LLM driven framework for detecting business logic vulnerabilities in smart contracts,” arXiv preprint arXiv:2602.03271, 2026.

[2] X. Hu, W. Y. Chan, Y. Shi, Q. Sun, W. C. Wang, C. Wu, H. Wang, and N. He, “An effective and cost-efficient agentic framework for Ethereum smart contract auditing,” arXiv preprint arXiv:2601.17833, 2026.

[3] S. Chang, C. Geng, H. Huang, R. Wang, Q. Li, and Y. Zhang, “Code Speak: I’m proving smart contract vulnerability detection via LLM-assisted code analysis,” Journal of Systems and Software, vol. 231, Art. no. 112635, 2026.

[4] H. Ding, Y. Liu, X. Piao, H. Song, and Z. Ji, “Smart Guard: An LLM-enhanced framework for smart contract vulnerability detection,” Expert Systems with Applications, vol. 269, Art. no. 126479, 2025.

[5] M. Bresil, P. W. C. Prasad, M. S. Sayeed, and U. A. Bukar, “Deep learning-based vulnerability detection solutions in smart contracts: A comparative and meta-analysis of existing approaches,” IEEE Access, vol. 13, pp. 28894–28919, 2025.

[6] J. Torres, M. Steichen, and R. State, “The art of the scam: Demystifying honeypots in Ethereum smart contracts,” in Proceedings of the USENIX Security Symposium, 2020.

[7] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in Proceedings of IEEE/ACM ICSE Workshops, 2019.

[8] K. Brent et al., “Vandal: A scalable security analysis framework for smart contracts,” arXiv:1809.03981, 2019.

[9] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in Proceedings of ACSAC, 2018.

[10] S. Tikhomirov et al., “SmartCheck: Static analysis of Ethereum smart contracts,” in Proceedings of IEEE ICST, 2018.

[11] P. Tsankov et al., “Securify: Practical security analysis of smart contracts,” in Proceedings of ACM CCS, 2018.

[12] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on Ethereum smart contracts,” in Proceedings of Principles of Security and Trust, 2017.

[13] T. Chen et al., “Under-optimized smart contracts devour your money,” in Proceedings of IEEE ICSE, 2017.

[14] L. Luu et al., “Making smart contracts smarter,” in Proceedings of ACM CCS, 2016.

[15] C. Szegedy et al., “Intriguing properties of neural networks,” in Proceedings of ICLR, 2014.

[16] I. Goodfellow et al., “Explaining and harnessing adversarial examples,” in Proceedings of ICLR, 2015.

[17] N. Papernot et al., “Practical black-box attacks against machine learning,” in Proceedings of ACM AsiaCCS, 2017.

[18] B. Biggio and F. Roli, “Wild patterns: Ten years after the rise of adversarial machine learning,” Pattern Recognition, 2018.

[19] N. Bn, D. E. Geetha, and R. G, “Parametric and non-parametric analysis on metaheuristic based event recommendation system,” in Proceedings of IEEE CISCON, 2025.

[20] N. B. Nithya, M. G. Parameshwari, G. H. Leela, G. S. Neeli, A. Awasthi, C. L. N. Deepika, and D. P. Kumar, “Neuro-Theological AI: Machine learning models for simulating mystical states of consciousness,” J. Data Sci., vol. 19, no. S5, pp. 871–878, 2026.

[21] N. Bn, S. B. Murthy, and S. Ds, “Improved quantum neural network for intrusion detection and Blowfish for data security,” in Proceedings of IEEE CISCON, 2025.

[22] G. Wood, “Ethereum: A secure decentralized generalized transaction ledger,” Ethereum Yellow Paper, 2014.

[23] V. Buterin, “A next-generation smart contract and decentralized application plat form,” Ethereum White Paper, 2014.

[24] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[25] ConsenSys Diligence, “Mythril: Security analysis tool for EVM bytecode,” 2024.