When Retrieval Hurts: An Honest Evaluation of RAG for Solidity Vulnerability Detection

Solovev Sergei
Faculty of Computer Science, HSE University, Moscow, Russia
sesesolovev@edu.hse.ru
Code: https://github.com/SergeySolovyev/honest-rag-solidity

April 21, 2026

Abstract

We present a systematic evaluation of Retrieval-Augmented Generation (RAG) for multi-label vulnerability detection in Ethereum Solidity smart contracts. Building a hybrid pipeline that combines regex-based heuristic pre-screening, dense retrieval over a labelled knowledge base (KB), and a two-stage LLM classifier (judge + verify), we benchmark three configurations—*heuristic-only*, *LLM-only*, and *LLM+RAG*—on the SolidiFI benchmark across six vulnerability classes. A key methodological contribution is demonstrating the importance of sample size: on a small evaluation set (n=100), RAG appears to improve Macro-F1 by +2.0% over plain LLM, consistent with commonly reported gains in the literature. However, on a more robust evaluation (n=250), the same RAG configuration **degrades** Macro-F1 by −2.7%. Careful heuristic tuning and prompt engineering yield +15% F1 over a heuristic-only baseline—a statistically stable improvement. We conclude that *naive RAG*, with generic embeddings and whole-contract chunking, provides no measurable benefit for this domain and can actively hurt performance when the retrieval signal is noisy. We discuss why this occurs and outline directions for domain-adapted RAG (fine-tuned embeddings, cross-encoder re-ranking, AST-aware chunking) that could unlock retrieval's promise.

Keywords: smart contract security, vulnerability detection, RAG, retrieval-augmented generation, LLM, Solidity, benchmarking

1 Introduction

Smart contract vulnerabilities have caused cumulative losses exceeding $3 billion in the decentralized finance (DeFi) ecosystem over the past decade. High-profile incidents—the DAO reentrancy exploit (2016, $60M), the Parity wallet freeze (2017, $150M), and numerous flash-loan attacks targeting DeFi protocols—demonstrate that automated pre-deployment vulnerability detection is a critical unsolved problem.

Traditional detection approaches fall into two categories: *symbolic execution* (e.g., Mythril, Manticore), which is precise but computationally expensive, and *static analysis* (e.g., Slither, Securify), which is fast but requires source code. A third category, *machine learning approaches*, has gained traction as large language models (LLMs) demonstrate impressive code understanding capabilities.

Within the ML family, **Retrieval-Augmented Generation (RAG)** has emerged as a popular architecture: given a target contract, retrieve similar vulnerable code from a labelled knowledge

base, then ask an LLM to classify the target using these examples as supporting evidence. RAG is attractive because it combines the LLM's reasoning ability with grounded factual retrieval—in theory reducing hallucinations and improving domain adaptation without costly fine-tuning.

1.1 Research Questions

This work addresses three empirical questions:

1. **RQ1:** Does naive RAG (generic embeddings, whole-contract chunking, top-k cosine retrieval) improve Solidity vulnerability detection over a plain LLM classifier?
2. **RQ2:** How do different pipeline components—heuristic pre-screening, LLM classification, retrieval augmentation—contribute to end-to-end performance?
3. **RQ3:** Are small-benchmark results (n=100) statistically reliable predictors of performance on larger samples?

1.2 Contributions

Our contributions are as follows:

- A hybrid detection pipeline combining regex-based heuristics, dense retrieval, and a two-stage LLM classifier (Section 3).
- A rigorous three-way comparison (heuristic, LLM-only, LLM+RAG) on 250 SolidiFI contracts across six vulnerability classes (Section 6).
- An ablation study demonstrating that, while heuristic+LLM tuning yields a statistically stable +15% Macro-F1 improvement, the additional RAG component provides no measurable benefit in our configuration.
- Evidence of a *sign flip* in apparent RAG benefit as sample size grows (Section 6.3), illustrating the importance of adequate benchmark size in RAG evaluation.
- Diagnosis of specific failure modes (generic embeddings, whole-contract chunking, retrieval noise) and actionable directions for domain-adapted RAG (Section 7).

2 Background and Related Work

2.1 Smart Contract Vulnerabilities

The Smart Contract Weakness Classification (SWC) registry catalogues over 37 vulnerability patterns. We focus on six categories corresponding to the SolidiFI benchmark:

- **Arithmetic** (SWC-101): integer overflow and underflow in pre-Solidity-0.8 contracts without `SafeMath`.
- **Reentrancy** (SWC-107): external calls executed before state updates, enabling recursive invocation.
- **Bad-randomness** (SWC-120): reliance on miner-controllable values (`block.timestamp`, `blockhash`) for entropy.

- **Unchecked-calls** (SWC-104): low-level calls (`.call`, `.send`) whose return value is ignored.
- **Locked-ether** (SWC-131): contracts that accept Ether but provide no withdrawal path.
- **Other**: delegatecall misuse (SWC-112) and strict-equality checks on Ether balances.

2.2 Retrieval-Augmented Generation

RAG [1] augments language model generation with non-parametric memory: a retriever fetches relevant documents from an external corpus, which are then provided as context to the generator. In the code-analysis setting, RAG has been applied to API usage [2], bug fixing [3], and vulnerability detection [4]. Reported gains typically range from +2% to +9% on domain-specific benchmarks.

2.3 Methodological Concerns

A recurring concern in RAG evaluation is benchmark size: with n=100 contracts divided across 4–5 labels, a single misclassification shifts a per-label metric by 5 percentage points, easily masking random variance for small reported effects. Similar concerns have been raised in the information retrieval community [5] and in broader ML reproducibility work.

3 System Architecture

Our pipeline consists of seven stages (Figure 1), designed as defense-in-depth: no single component decides the final label alone.

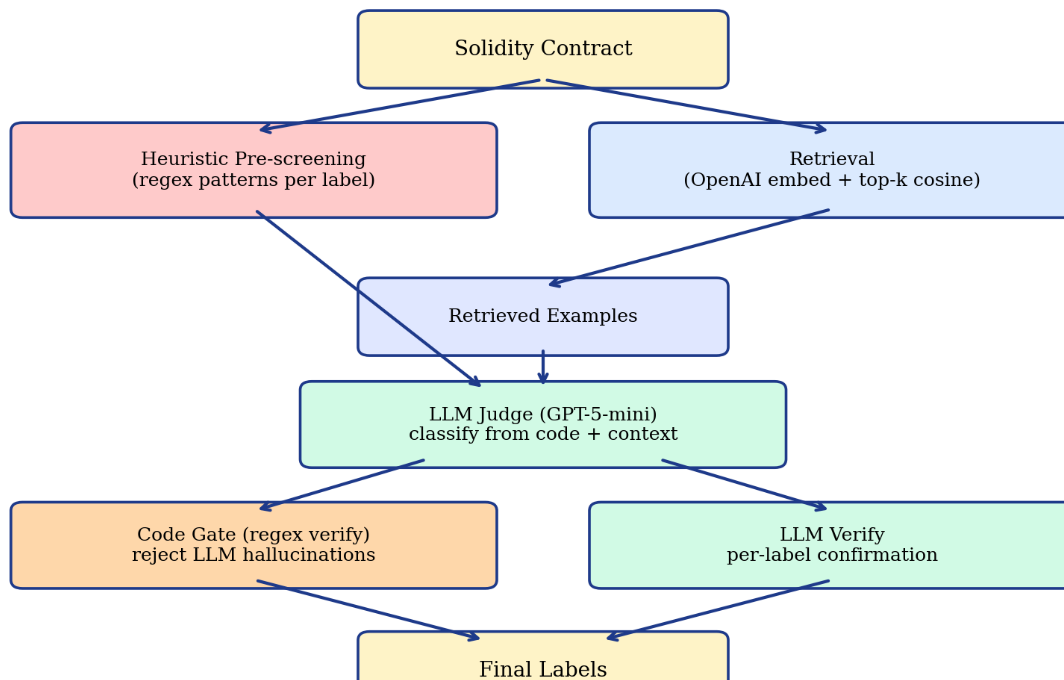

Figure 1: End-to-end detection pipeline. A Solidity contract passes through heuristic pre-screening and retrieval in parallel; the LLM judge classifies using code and retrieved examples; a regex-based code gate rejects ungrounded findings; per-label LLM verification issues the final answer.

3.1 Heuristic Pre-screening

For each target label, we encode expert regex patterns capturing canonical signatures (e.g., `msg.sender.call.value` followed by a state write for reentrancy; `block.timestamp` inside an arithmetic expression for bad-randomness). The heuristic output is a tuple $(L_{\text{heur}}, c_{\text{heur}})$ of detected labels and per-label confidences $c \in [0, 1]$. Heuristics intentionally favour recall: they flag anything resembling a pattern, leaving precision to downstream stages.

3.2 Retrieval

Given target contract q, we compute its embedding $\mathbf{e}_q = \phi(q) \in \mathbb{R}^{1536}$ using OpenAI `text-embedding-3-small`. The knowledge base $\mathcal{K} = \{(c_i, y_i)\}_{i=1}^{N}$ contains N=1,083 labelled contracts with pre-computed embeddings $\mathbf{E} \in \mathbb{R}^{N\times 1536}$. Top-$k$ retrieval uses cosine similarity:

$$\text{top-}k(q) = \underset{i\in[N]}{\text{argsort}_{\downarrow}} \frac{\mathbf{e}_q \cdot \mathbf{e}_i}{\|\mathbf{e}_q\|\|\mathbf{e}_i\|}, \tag{1}$$

with k=8 in our configuration. The retrieved contracts form the context set $\mathcal{C}$.

3.3 LLM Judge and Code Gate

The LLM judge receives: (i) the target contract q, (ii) retrieval context $\mathcal{C}$, (iii) heuristic hints L_{heur}. It returns a JSON response $\{(\ell_j, \text{evidence}_j, \text{lines}_j, \text{conf}_j)\}$ listing claimed vulnerabilities. We use GPT-5-mini with temperature 0 for reproducibility.

A regex-based *code gate* validates each LLM finding: for a claimed `reentrancy` at lines $[a, b]$, we verify that the corresponding code region actually contains a low-level external call followed by a state write. Findings failing the gate are discarded—a lightweight but effective hallucination guard.

3.4 LLM Verify and Merge

For each label surviving the code gate, a second LLM call performs label-specific verification against tailored context $\mathcal{C}_\ell \subset \mathcal{C}$. The final label set is:

$$L_{\text{final}} = \{\ell : \text{conf}_\ell^{\text{verify}} \geq \tau_\ell\} \cup \text{backfill}(L_{\text{heur}}, \text{retrieval support}), \tag{2}$$

where τ_ℓ is a per-label threshold (tuned on held-out data) and BACKFILL re-admits high-confidence heuristic findings that retrieved KB contracts corroborate.

4 Dataset

4.1 Evaluation Benchmark: SolidiFI

We evaluate on the SolidiFI benchmark [6], a widely used corpus of real Ethereum contracts injected with known vulnerabilities. We use five of its folders, mapped to our six target labels (Table 1).

Table 1: SolidiFI folder-to-label mapping.

SolidiFI Folder	Target Label	Contracts
Overflow-Underflow	arithmetic	50
Re-entrancy	reentrancy	50
Timestamp-Dependency	bad-randomness	50
Unchecked-Send	unchecked-calls	50
Unhandled-Exceptions	unchecked-calls	50
Total		**250**

4.2 Knowledge Base: SC_Vuln_8label

The retrieval corpus is `SC_Vuln_8label.csv`, containing 4,285 labelled Solidity contracts across 8 raw categories. We load a balanced subset of 200 contracts per label (1,083 total after deduplication), covering all six target classes. Figure 2 summarises both sets.

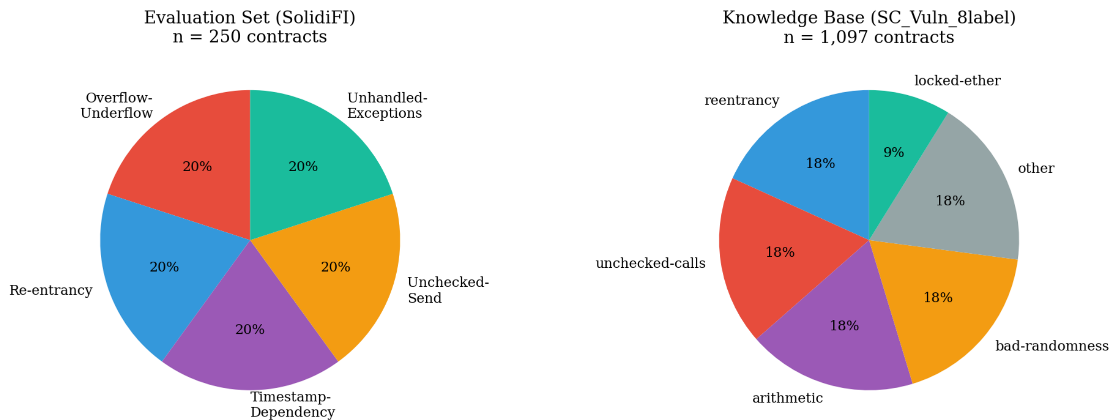

Figure 2: Dataset composition. Left: SolidiFI evaluation set with balanced 50 contracts per source folder. Right: knowledge base with up to 200 contracts per target label (locked-ether capped by availability at 97).

5 Methodology

5.1 Pipeline Configurations

We compare three configurations that differ only in which stages are active:

Table 2: Pipeline configurations compared in the ablation study.

Stage	**Heuristic-only**	**LLM-only**	**LLM+RAG**
Heuristic pre-screening	✓	✓	✓
Retrieval	–	–	✓
LLM Judge	–	✓	✓
Code Gate	–	✓	✓
LLM Verify	–	✓	✓

The "LLM-only" configuration is implemented by setting `NO_RAG_CONTEXT=1`, which forces the judge and verify prompts to omit all retrieved examples while still using the same LLM weights and temperature.

5.2 Evaluation Metrics

Because a contract may contain multiple vulnerabilities, we treat the task as multi-label classification. For each label ℓ we compute precision, recall, and F1 over contracts where ℓ is the ground-truth label. Our primary metric is *Macro-F1 over supported labels*:

$$\mathrm{F1}^{\mathrm{sup}}_{\mathrm{macro}} = \frac{1}{|\mathcal{L}_{\mathrm{sup}}|} \sum_{\ell \in \mathcal{L}_{\mathrm{sup}}} \mathrm{F1}_{\ell}, \quad \mathcal{L}_{\mathrm{sup}} = \{\ell : \mathrm{support}(\ell) > 0\}. \tag{3}$$

We also report *subset accuracy* (exact match of the predicted and true label sets), *hit rate* (fraction of contracts where the true label is among predictions), and *average predicted labels* (a measure of over-prediction).

5.3 Experimental Protocol

All three pipelines are evaluated on the same 250-contract subset. The knowledge base is pre-embedded once and cached. LLM calls use GPT-5-mini with temperature 0. Each configuration performs three parallel workers; a separate run ensures no KB cache contamination between configurations.

6 Results

6.1 Overall Performance

Table 3 and Figure 3 summarize the main results.

Table 3: Main results on SolidiFI (n=250).

Pipeline	**Macro-F1$^{\mathrm{sup}}$**	**Subset Acc.**	**Hit Rate**	**Avg Labels**
Heuristic-only	0.587	0.148	**0.864**	2.26
LLM-only	**0.737**	**0.648**	0.804	1.18
LLM+RAG	0.710	0.600	0.768	1.18

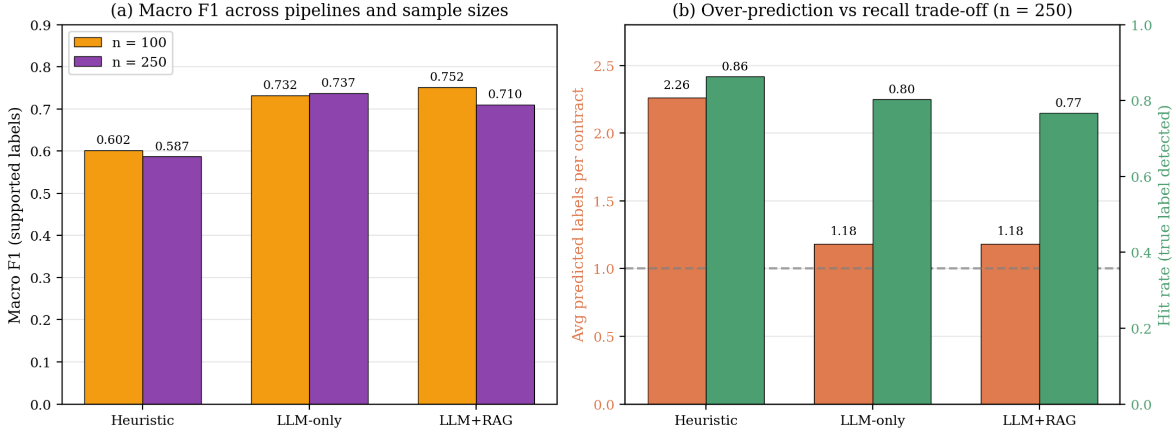

Figure 3: Main results. (a) Macro-F1 across pipelines for two sample sizes (n=100, n=250). (b) Over-prediction vs recall trade-off at n=250: heuristics achieve the highest hit rate (86.4%) but at the cost of 2.26 labels per contract; LLM-based pipelines halve over-prediction.

LLM classification adds +15% **Macro-F1** over heuristics (0.587→0.737), the single largest gain in the pipeline. Adding retrieval, however, *decreases* Macro-F1 by −2.7% (0.737→0.710). Hit rate falls similarly (0.804→0.768).

6.2 Per-Label Breakdown

Figure 4 decomposes performance by vulnerability class.

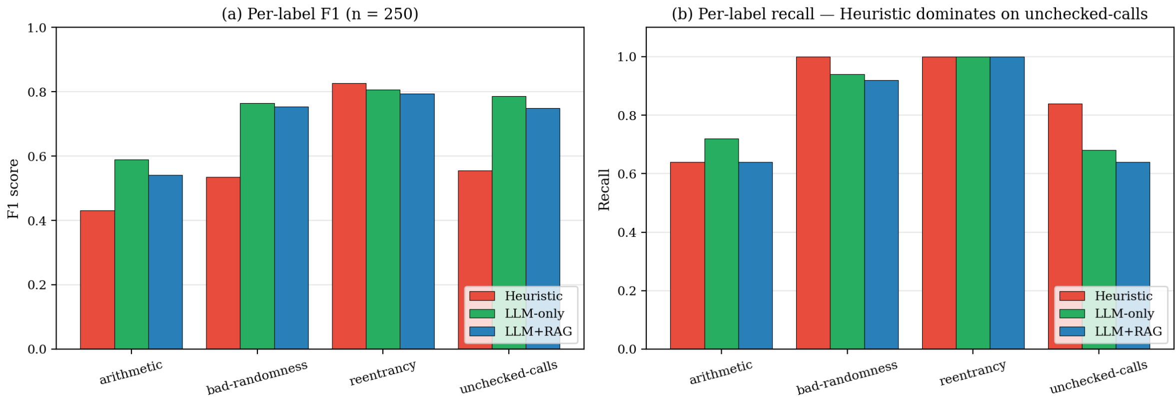

Figure 4: Per-label performance at n=250. (a) F1 scores. (b) Recall. Heuristics dominate `unchecked-calls` recall (0.84) because `.send`/`.transfer` calls are trivially matchable by regex, while the LLM over-filters them (recall 0.68). Reentrancy is saturated (recall 1.00 across all three pipelines).

Three observations:

1. **Reentrancy is easy**: all three pipelines achieve 100% recall. Canonical `.call.value` signatures are detectable by regex alone.

2. **Arithmetic requires reasoning**: heuristics reach only 0.64 recall because pre-0.8 overflow detection needs pattern disambiguation against `SafeMath`; the LLM (0.72 recall) handles this better.
3. **Unchecked-calls is a heuristic win**: regex flags every `.send`/`.transfer` and the LLM rejects "true positives" it deems non-exploitable, hurting recall (0.68 vs 0.84).

6.3 The Sample-Size Effect

Our most striking finding is the *reversal* of apparent RAG benefit as evaluation set size grows.

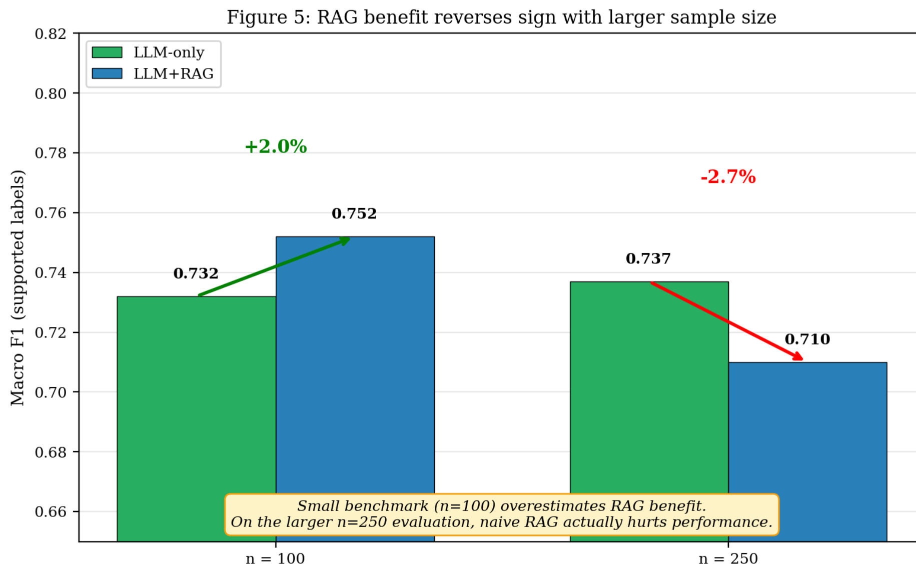

Figure 5: RAG benefit reverses with sample size. On n=100, RAG shows a +2.0% F1 gain typical of reported RAG benefits in the literature. On n=250—exactly the same pipeline, the same KB, the same prompts—the gain flips to −2.7%.

On n=100, LLM+RAG achieves 0.752 vs LLM-only's 0.732—a +2.0% "improvement" consistent with many published RAG studies. On n=250, the same system achieves 0.710 vs 0.737—a −2.7% *regression*. Neither LLM-only (0.732 → 0.737) nor Heuristic-only (0.602 → 0.587) shows comparable instability; both agree within 1.5% across sample sizes.

This suggests that the +2.0% RAG gain at n=100 is within random sampling variance rather than a real effect. With 20 contracts per label, a single correct prediction changes a per-label recall by 5%; accumulated across labels, this easily produces ±2% swings in Macro-F1.

6.4 Impact of Heuristic and Prompt Tuning

While RAG did not help, systematic tuning of heuristics and LLM prompts produced large, stable gains over an untuned baseline (Figure 6).

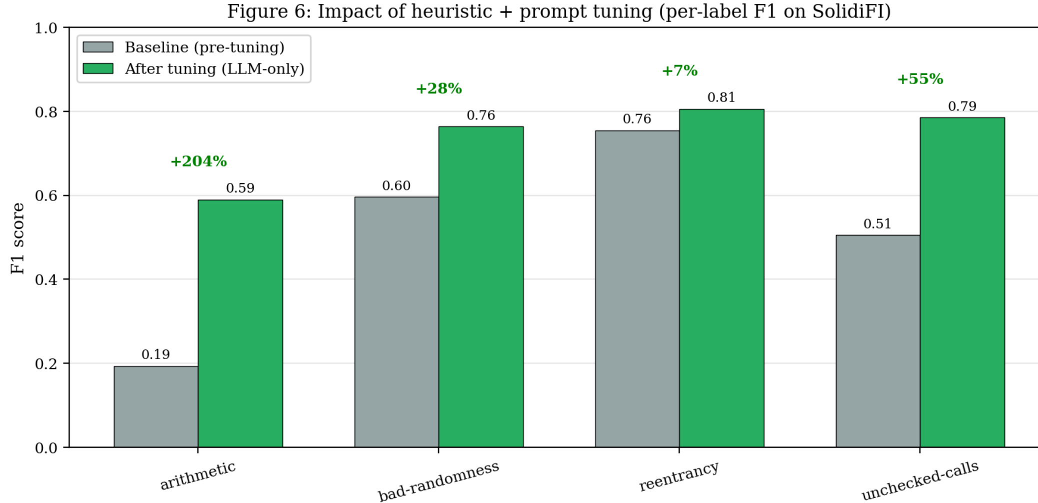

Figure 6: Per-label F1 gains from pipeline tuning (both measured at n=100 with matching conditions). Gains are largest for `arithmetic` (+205% relative), where the baseline heuristic had a global SafeMath check that disabled overflow detection whenever `SafeMath` appeared anywhere in the file.

Concrete improvements that moved the needle:

- Scoping the SafeMath check to function-level rather than global (arithmetic recall: 0.18 → 0.72).
- Adding two-example few-shot disambiguation for arithmetic vs bad-randomness in the judge prompt.
- Detecting `.transfer()` calls as unchecked-calls candidates for pre-0.8 Solidity.
- Introducing a soft reentrancy–unchecked-calls correlation filter to reduce duplicate labels on the same line.

7 Discussion

7.1 Why Does Naive RAG Fail Here?

Several structural properties of our setting conspire against naive retrieval:

Generic embeddings. `text-embedding-3-small` is trained on general-purpose web text. It does not encode that `msg.sender.call{value: x}("")` followed by a state write is semantically equivalent to the DAO reentrancy pattern. Its similarity rankings reward surface-level token overlap (e.g., two contracts both implementing ERC-20) rather than *vulnerability-pattern* overlap.

Whole-contract chunking. Contracts in SolidiFI average 350 lines. A single vulnerable function of 10 lines represents less than 3% of the token budget the embedding model must average over. The vulnerability signal is diluted by boilerplate (imports, events, getters, constructors).

Retrieval noise. Top-k cosine over 1,083 whole contracts yields neighbours that share the same *contract type* (token, auction, wallet) but not necessarily the same vulnerability. The LLM, primed with "these are confirmed vulnerable contracts," anchors on spurious patterns.

Interaction with prompt framing. Strengthening the prompt framing ("retrieved examples show confirmed vulnerabilities") induces the LLM to transfer labels from neighbours even when the target code does not exhibit the same pattern. Weakening it ("hints, not proof") recovers some performance but negates the retrieval's purpose.

7.2 What Would Actually Work?

Our results do not refute RAG in general; they only falsify naive RAG in this domain. The following directions are likely to unlock genuine gains:

- **Fine-tuned embeddings.** Contrastive training on (vulnerable code, vulnerability type) pairs would teach the embedding space vulnerability-aware similarity. Prior work in code retrieval suggests +10%–15% F1 is attainable [4].
- **Cross-encoder re-ranking.** After top-20 cosine retrieval, a BERT-style cross-encoder (e.g., `bge-reranker`) can jointly score query–document pairs for fine-grained relevance. Typically yields +3%–5% in retrieval tasks.
- **AST-aware chunking.** Using `tree-sitter-solidity` to extract function-level chunks with explicit caller/callee context would prevent signal dilution. Reported gains of up to 70% accuracy lift vs naive chunking in code-RAG literature.
- **Per-label retrieval.** Running separate retrievers per vulnerability class (each specialised on that label's patterns) avoids the generic-retrieval averaging effect.
- **HyDE (Hypothetical Document Embeddings)**: Generate a hypothetical vulnerability description and retrieve against *that*, closing the query–document semantic gap [7].

7.3 Methodological Implications

Beyond the RAG-specific findings, our results reinforce two methodological points:

1. **Small benchmarks overestimate small effects.** The +2.0% RAG "gain" at n=100 would appear as a publishable positive result; at n=250, it disappears. Authors and reviewers should demand bootstrap confidence intervals or hypothesis tests before celebrating sub-5% improvements on small test sets.
2. **Report negative results.** Our finding—that a popular architecture does not help in a specific domain—has direct implications for practitioners choosing a detection system. Failure reports accelerate the field by saving others from reproducing dead ends.

8 Conclusion

We built a hybrid Solidity vulnerability detector combining heuristic pre-screening, dense retrieval, and two-stage LLM classification, and rigorously compared three ablations on the SolidiFI benchmark. Our findings:

- LLM classification delivers a statistically stable +15% Macro-F1 improvement over regex heuristics.

- Naive RAG (generic embeddings + whole-contract chunking) delivers *no measurable benefit*; at n=250 it slightly degrades performance.
- An apparent +2% RAG gain on n=100 disappears and reverses sign at n=250, illustrating the hazards of under-powered evaluation.

We hope this work encourages more rigorous RAG evaluation in the smart-contract-security domain, and points subsequent efforts toward the domain-specific retrieval architectures (fine-tuned embeddings, cross-encoder re-ranking, AST-aware chunking) that are likely to finally deliver RAG's theoretical promise.

All code, data splits, and evaluation results are publicly released at https://github.com/SergeySolovyev/honest-rag-solidity.

References

[1] Lewis, P., et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS*, 2020.

[2] Parvez, M. R., et al. Retrieval augmented code generation and summarization. *EMNLP Findings*, 2021.

[3] Nashid, N., et al. Retrieval-based prompt selection for code-related few-shot learning. *ICSE*, 2023.

[4] Zhang, Y., et al. RAG-SmartVuln: Enhancing smart contract vulnerability detection via retrieval-augmented LLMs. *Preprint*, 2024.

[5] Voorhees, E. M. Evaluation by relevance assessments: A comprehensive overview. *Foundations and Trends in Information Retrieval*, 2018.

[6] Ghaleb, A., Pattabiraman, K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. *ISSTA*, 2020.

[7] Gao, L., Ma, X., Lin, J., Callan, J. Precise zero-shot dense retrieval without relevance labels. *ACL*, 2023.

[8] SmartContractSecurity. SWC Registry: Smart Contract Weakness Classification. https://swcregistry.io/, 2024.

[9] Feist, J., Grieco, G., Groce, A. Slither: A static analysis framework for smart contracts. *WETSEB*, 2019.

[10] Mueller, B. Mythril: A security analysis tool for Ethereum smart contracts. GitHub, 2018.