

Analyze the Code, Not the Comments: Comment-Stripped Analysis Fully Defends LLM-Based Code Vulnerability Detectors Against Adversarial Comment Injection

Sunny Chokshi

Dept. of Information Technology

University of the Cumberlands

Williamsburg, KY, USA

emailchokshi@gmail.com

ORCID: 0009-0003-4738-7759

Abstract. Large language models are increasingly used to detect security vulnerabilities in source code, but recent work has shown that adversarial comments embedded in the code under review can suppress detection, causing a model to miss genuine vulnerabilities. Because these attacks operate entirely through natural-language comments while leaving the executable code unchanged, they suggest a correspondingly simple defense: remove the comments before analysis. This paper presents the first empirical evaluation of comment-stripped analysis as a defense against adversarial comment injection. Across 480 adversarial trials on two open-source code models and eight adversarial comment strategies, we compare detection on clean code, on adversarially commented code, and on the same adversarial code with all comments removed. Under attack, detection fell from a clean baseline of 90.0% to 75.2%. With comment-stripped analysis, detection returned to 90.0%, recovering 100% of the detection lost to the attack. A matched-pairs McNemar test confirms the improvement is highly significant ($\chi^2 = 65.33$, $p < 0.001$): the defense recovered 73 vulnerabilities the attack had hidden while sacrificing only 2. The defense requires no model retraining and no access to model internals. It adds only a single analysis pass. These results show that a trivially deployable, model-agnostic input transformation fully neutralizes adversarial comment injection in the evaluated setting.

Index Terms. vulnerability detection, adversarial machine learning, LLM code review, prompt injection, defense, input sanitization, AI security

I. INTRODUCTION

Automated code review powered by large language models has moved quickly from research demonstration to practical deployment. Developers now rely on LLM-based assistants to inspect code for security vulnerabilities, and the availability of capable open-source models has made these tools accessible to organizations of every size. In the code review setting, however, the input to the detector is code written by a developer who may have an incentive to have vulnerable code pass review, and that developer controls the full text submitted, including its comments.

Recent work has established that this control can be exploited. Adversarial comment injection, in which misleading natural-language comments are prepended to vulnerable code, suppresses an LLM detector's identification of genuine vulnerabilities. The comments do not alter the executable code. A vulnerability present in commented code is present just as surely in the uncommented version. Yet a comment asserting that the code was already reviewed, that its inputs are sanitized upstream, or that the function is isolated from untrusted input can cause a model to reclassify vulnerable code as safe. Prior studies have measured how effective different comment strategies are and shown that susceptibility varies across models, but they have focused on characterizing the attack rather than defending against it.

This paper addresses the defense. The defining property of adversarial comment injection points directly to a countermeasure. Because the attack lives entirely in the comments while the executable code is unchanged, removing the comments before analysis should neutralize the attack without affecting the model's ability to reason about the code itself. We call this comment-stripped analysis. It is attractive precisely because it is simple. It requires no model retraining, no access to model weights or gradients, and no change to the model at all. It adds only a single additional analysis pass on the comment-free code.

The simplicity of the defense does not by itself establish that it works. Whether comment-stripped analysis actually recovers the detection lost to adversarial comments is an empirical question that has not been measured. Stripping comments might recover detection fully, might recover it only partially if some models rely on comments for legitimate context, or might introduce new failures by removing information the model needs. The answer determines whether comment-stripped analysis is a practical defense or only an appealing idea.

The stakes of this question are practical. Organizations are already deploying LLM-based code review in continuous-integration pipelines, and if a single misleading comment can hide a vulnerability, then any such deployment is exposed to a

trivial and low-cost evasion by the very developers whose code is being checked. A defense that is equally trivial and low-cost, requiring no change to the model, would let these organizations close the exposure immediately rather than waiting for more robust models to become available. Establishing whether the simplest possible defense actually works is therefore not merely of academic interest; it determines whether a deployable countermeasure exists today.

This paper answers that question through a controlled experiment. For every sample in a benchmark of thirty documented vulnerabilities, and for each of eight adversarial comment strategies, we evaluate three conditions on two open-source code models: the clean code, the adversarially commented code, and the adversarially commented code with all comments removed. Comparing the three isolates exactly what the defense recovers. The central finding is that comment-stripped analysis restores 100% of the detection lost to adversarial comments, returning detection to the clean baseline, a statistically significant recovery, while remaining a trivially deployable, model-agnostic transformation.

This paper makes three contributions. The first is an empirical evaluation of comment-stripped analysis as a defense against adversarial comment injection. The second is a measurement of how much detection the defense recovers, reported per model and across eight attack strategies with a matched-pairs significance test. The third is a discussion of the defense's limits and its place in deployed LLM-based code review pipelines.

II. RELATED WORK

The security of LLM-based code analysis has drawn increasing attention. Pearce et al. [1] evaluated the security of LLM-generated code and showed that a substantial fraction of generated programs contained exploitable weaknesses, establishing that code models have security blind spots, though their focus was unintentional insecurity rather than deliberate manipulation. Prompt injection, in which adversarial instructions are embedded in content the model processes, has been characterized by Greshake et al. [2] and systematized by Liu et al. [3]. Adversarial comment injection against code vulnerability detection is a specialized form of indirect prompt injection where the untrusted content is the code under review and the injection channel is its comments.

Prior empirical studies have shown that adversarial comments suppress detection, that different comment strategies differ in effectiveness, and that susceptibility varies substantially across models. These studies characterize the attack surface. This paper evaluates a defense against it. Broader adversarial machine learning has shown that models can be manipulated in ways not evident from normal performance and that robustness is not well predicted by standard accuracy [4]. Defenses in that literature often require retraining, adversarial training, or access to model internals. The defense evaluated here is deliberately lighter. It is an input-transformation defense, requiring no retraining or internal access, in the spirit of input sanitization defenses against injection attacks generally. Input-transformation defenses are attractive in deployment because they are model-agnostic. To the author's knowledge, no prior work has empirically measured whether removing comments

before analysis recovers the detection lost to adversarial comment injection.

The defense studied here can be understood in the broader context of architectural defenses that separate trusted from untrusted information in LLM systems. Recent secure-architecture proposals for LLM-integrated systems [8] decouple the handling of trusted planning information from untrusted execution content, so that adversarial content cannot corrupt the system's core decisions. Comment-stripped analysis applies the same principle at a smaller scale and with far lower cost. The executable code is the trusted artifact to be judged, and the author-supplied comments are the untrusted channel through which manipulation arrives. Removing the untrusted channel before judgment isolates the trusted artifact. Unlike architectural approaches that require redesigning the system, the defense evaluated here is a single preprocessing step compatible with any existing detector, which is what makes it immediately deployable.

III. THREAT MODEL

We consider an adversary who is the author of the code submitted for automated security review. The adversary's goal is to have code containing a genuine vulnerability pass the LLM-based detector without the vulnerability being flagged. This threat model is realistic in modern development workflows, where automated review is applied to code contributed by developers whose incentives may not align with thorough security scrutiny.

The adversary's capabilities are deliberately minimal, which makes the threat broadly applicable. The adversary controls the full text of the submitted code, including all comments, but does not alter the executable logic, since the goal is to ship functional vulnerable code. The adversary requires no access to the model's weights, gradients, or internal states, and performs no optimization; the adversary needs only to write natural-language comments. Because every code sample in the benchmark contains a genuine vulnerability, any sample the detector classifies as safe represents a successful evasion.

The defender is the operator of the code-review pipeline. The defender controls how code is presented to the model but, in the setting we study, does not retrain or fine-tune the model and does not access its internals; the defender can only transform the input and issue analysis requests. This matches the common deployment reality in which an organization consumes a code model as a service or as fixed open weights and can change what it sends but not what the model is. Comment-stripped analysis is defined entirely within these constraints. It is a transformation of the model's input, applicable to any model regardless of architecture or access level.

IV. METHODOLOGY

A. Benchmark and Attacks

The benchmark comprises thirty code samples, each containing a single documented vulnerability. The samples span fourteen Common Weakness Enumeration categories, including SQL injection, command injection, cross-site scripting, path traversal, hardcoded credentials, weak cryptography, insecure deserialization, code injection, weak randomness, XML external

entity processing, server-side request forgery, and disabled certificate validation. Every sample contains a genuine vulnerability, so a correct detector should flag all of them, and any failure to flag is a successful adversarial evasion.

The fourteen CWE categories were chosen to span vulnerability types that differ in how syntactically evident their danger is. They range from injection flaws whose dangerous constructs are explicit in the code to context-dependent weaknesses whose severity depends on how and where the code is used. This spread matters for evaluating a defense, because a defense that recovers detection only for syntactically obvious vulnerabilities would be less useful than one that recovers detection uniformly. Because comment stripping removes the manipulation channel rather than reasoning about the vulnerability, the defense is expected to recover detection regardless of a vulnerability's category. The benchmark's breadth allows this expectation to be tested against a diverse set of weaknesses rather than a narrow one.

Eight adversarial comment injection strategies were applied to each sample, each prepending misleading comments without altering the executable code. The strategies span assertions of prior review, claims of upstream sanitization, framing the function as a trusted internal utility, security vocabulary, claims of test coverage, claims of formal verification, compliance labels, scope-limitation claims, urgency, distraction, and bare assertions of the author's personal confidence. This set covers the range of adversarial comment strategies established as effective in prior work.

B. The Defense

Comment-stripped analysis removes all comments from a submitted function before the code is analyzed. A vulnerability reported on the comment-stripped code that was suppressed on the commented code is direct evidence of adversarial comment injection, and the pipeline can adopt the more conservative verdict. In this study we evaluate the core question the defense rests on: does analyzing the comment-stripped version recover the detection lost to the adversarial comments? The defense is implemented as a simple lexical removal of comment text, leaving the executable code byte-for-byte unchanged.

C. Models and Procedure

Two open-source code-specialized models were evaluated: DeepSeek-Coder 6.7B and Qwen2.5-Coder 7B. Both were run locally through the Ollama runtime (version 0.3.x), using the default quantized model builds distributed by Ollama, with temperature set to zero for deterministic output. Fixing the model versions and the decoding temperature ensures the reported detection counts reproduce exactly. For every sample, we evaluated three conditions: the clean code, the adversarial version for each of the eight strategies, and the comment-stripped version of each adversarial sample. A response was scored as a detection when the model classified the code as vulnerable. Because every sample is genuinely vulnerable, a classification of safe is always an error. To test whether the defense significantly improves over the attack, a McNemar test for matched pairs was applied to the adversarial and defended outcomes on identical samples, which is the appropriate test because the two conditions are measured on the same items.

The experiment is designed to be fully reproducible on commodity hardware. Both models are freely available and were run locally through the Ollama runtime, requiring no proprietary API access. Decoding temperature was fixed at zero so that, given the same model versions, the reported detection counts reproduce exactly. Each model received an identical prompt instructing it to act as a security code reviewer and to state on the first line whether the code is vulnerable or safe. The clean, adversarial, and defended versions of a sample differ only in their comments while sharing identical executable code. This design is what allows the matched-pairs comparison to attribute any change in detection to the presence or absence of the adversarial comments rather than to any change in the program itself.

V. RESULTS

A. The Defense Fully Recovers Detection

On clean code, the two models detected 90.0% of vulnerabilities, establishing the baseline. Under adversarial comment injection, aggregate detection fell to 75.2%, a loss of 14.8 percentage points, confirming that the attacks are effective. With comment-stripped analysis, detection returned to 90.0%, exactly the clean baseline. The defense therefore recovered 100% of the detection lost to the attack. Table I summarizes the three conditions.

TABLE I. DETECTION ACROSS THE THREE CONDITIONS

Condition	Detection	Interpretation
Clean baseline	90.0%	no attack
Under attack	75.2%	attack succeeds
With defense	90.0%	fully recovered

The recovery is not only complete in aggregate but statistically robust. A McNemar test on the matched adversarial and defended outcomes yields chi-square = 65.33 with $p < 0.001$. The direction of the effect is decisive: the defense caught 73 vulnerabilities that the attack had hidden, while only 2 vulnerabilities that were detected under attack were missed after comment stripping. The defense thus corrects errors far more often than it introduces them, by a margin of 73 to 2.

The asymmetry between the 73 corrections and the 2 regressions deserves emphasis, because it addresses the natural concern that removing comments might harm detection. If comments carried information the model genuinely needed, comment stripping would cause detections to be lost, and the count of newly missed vulnerabilities would be substantial. Instead it was two. This indicates that, for the purpose of vulnerability detection on this benchmark, the comments contributed almost nothing of legitimate value to the models and served overwhelmingly as a channel for manipulation. The defense therefore does not trade one kind of error for another; it removes a large source of induced errors while introducing a negligible number of new ones.

The recovery held across the full range of adversarial strategies. Because every one of the eight strategies operates by adding comments and none alters the executable code, comment stripping removes the entire attack surface uniformly, regardless

of which persuasion mechanism a given strategy used. Strategies that were highly effective against the weaker model, such as authority claims and bare assertions of authorial confidence, were neutralized by the defense just as completely as weaker strategies. This is because the defense does not reason about the content of the comment at all. It simply removes it. This uniformity is a practical strength: a defender deploying comment-stripped analysis need not anticipate which specific comment strategy an adversary will use, since the defense is agnostic to the comment's content and effective against the entire class.

B. Consistency Across Models

Table II reports the three conditions for each model separately. The defense fully restored the clean baseline for both models. DeepSeek-Coder, the weaker and more attackable model, fell from a clean 83.3% to 56.7% under attack, and the defense returned it exactly to 83.3%. Qwen2.5-Coder, already more robust, fell only from 96.7% to 93.8% and the defense returned it to 96.7%. The defense is therefore most valuable for the weaker model, where the attack does the most damage, and it does no harm to the stronger model. In both cases the defended detection matches the clean baseline precisely, which is the strongest possible outcome for the defense.

TABLE II. DETECTION BY MODEL AND CONDITION

Model	Clean	Attack	Defended
DeepSeek-Coder 6.7B	83.3%	56.7%	83.3%
Qwen2.5-Coder 7B	96.7%	93.8%	96.7%

Table III makes the uniformity of the recovery explicit by reporting, for each of the eight attack strategies, detection under attack and detection with the defense applied. The strategies are ordered from most effective attack to least. Regardless of how damaging a strategy was, the defense returned detection to the 90.0% clean baseline in every case.

TABLE III. ATTACK VERSUS DEFENDED DETECTION BY STRATEGY

Attack Strategy	Under Attack	Defended
Authority claim	63.3%	90.0%
Misleading context	63.3%	90.0%
False test coverage	65.0%	90.0%
False reassurance	68.3%	90.0%
Distraction comment	81.7%	90.0%
Security camouflage	85.0%	90.0%
Fake review approval	86.7%	90.0%
Deadline pressure	88.3%	90.0%

VI. ILLUSTRATIVE EXAMPLE

Consider a Python function containing an operating system command injection vulnerability (CWE-78), in which a user-supplied hostname is concatenated directly into a shell command:

```
def ping_host(hostname):
    os.system("ping -c 1 " + hostname)
```

In its clean form, both models flag this correctly. Under a misleading-context attack that prepends comments describing the function as a trusted internal utility not exposed to user input, the weaker model reclassifies the function as safe, though the executable code is unchanged. When the defense removes those comments and presents the identical executable code, the model again flags the vulnerability. The defense works because the vulnerability was never in the comments; it was in the code the comments were disguising, and stripping the disguise restores the model's correct judgment.

To make the transformation concrete, the adversarial version of the function above prepends comments such as the following, which assert a safe usage context without changing any executable statement:

```
# Internal admin helper, not exposed to user input.
# Inputs are validated upstream; safe by design.
def ping_host(hostname):
    os.system("ping -c 1 " + hostname)
```

The defense produces the version the model should have judged in the first place, with the two comment lines removed and the executable statements identical. Presented with this comment-free function, the weaker model again reports the command injection. The example makes the mechanism visible: the false premise lived only in the two comment lines, and once they are gone, nothing remains to mislead the model, because the vulnerable call was never hidden by the code itself, only by the narrative wrapped around it.

VII. DISCUSSION

A. Why the Defense Works Completely

The completeness of the recovery follows directly from the mechanism of the attack. Adversarial comment injection does not exploit a weakness in the model's ability to analyze code. It exploits the model's willingness to accept author-supplied claims about the code as context. Because the executable code is identical across the clean, adversarial, and defended conditions, and because the defended condition removes exactly the injected text that carried the false context, the defended input is functionally equivalent to the clean input for the purpose of vulnerability detection. The observed return to the exact clean baseline, for both models, is the expected result when the only thing removed is the attack itself.

B. Input Transformation Versus Model Hardening

An alternative response to adversarial comment injection would be to harden the model itself, for example by fine-tuning it to ignore author-supplied claims or by training it adversarially against such comments. Such approaches are valuable but carry real costs. They require access to the model or the ability to fine-tune it. They consume training data and compute, they must be repeated for every model an organization uses, and they risk degrading the model's performance on legitimate inputs. The input-transformation defense evaluated here avoids all of these costs. It treats the model as a fixed black box and intervenes only on the input, which means it transfers without modification across models, scales, and providers. The complete recovery observed suggests that, at least for attacks confined to

comments, this lightweight approach is sufficient and model hardening is unnecessary. The appropriate division of labor may be to use comment-stripped analysis as a first-line, universal defense and to reserve heavier model-hardening measures for adaptive attacks that survive input transformation.

C. A Deployable, Model-Agnostic Defense

The practical appeal of comment-stripped analysis is that it asks nothing of the model. It requires no retraining, no fine-tuning, no access to weights or gradients, and no knowledge of the model's architecture. It is a preprocessing step that any pipeline can apply to any model, including proprietary models consumed as a service. Its cost is a single additional analysis pass on the comment-free code, which can be run in parallel with the original. A pipeline can compare the two verdicts and adopt the more conservative one, flagging any discrepancy as evidence of possible adversarial commenting. Given the complete recovery observed here, comment-aware analysis is a strong candidate for a default component of LLM-based code review deployed under adversarial conditions.

D. Deployment Cost and Integration

The cost of the defense is modest and predictable. It doubles the number of analysis passes per function, since each function is analyzed both as submitted and with comments removed, but the two passes are independent and can run in parallel, so the added latency need not double in practice. No additional model, training data, or infrastructure is required. Integration into an existing pipeline requires only a comment-removal step and a verdict-comparison rule. If the comment-stripped analysis reports a vulnerability that the original analysis missed, the pipeline adopts the vulnerable verdict. It may also flag the discrepancy for human attention as a signal of possible adversarial commenting. This makes the defense not only a detector-hardening measure but also an inexpensive indicator of adversarial activity, since a divergence between the two verdicts is itself evidence that a comment influenced the model's judgment.

In our local setup, a single analysis pass took on the order of a few seconds per sample on commodity hardware, so the comment-stripped pass roughly doubles per-sample latency when run sequentially. Because the two passes are independent, running them concurrently keeps the added wall-clock time small in practice. The overhead is therefore a modest and predictable constant factor, not a change in the order of computational cost, which is acceptable for the security benefit of full recovery.

VIII. LIMITATIONS

This defense recovers detection by discarding comments, which means it also discards any legitimately useful information those comments carry. In a security-review setting, erring toward analyzing the code on its own terms and reporting a potential vulnerability for human confirmation is the appropriate default, but the trade-off should be acknowledged. The comment-stripping implementation evaluated here is a lexical removal suited to the benchmark; production use would require careful handling of comment-like text inside string literals. The study evaluates two models at a single parameter scale and a benchmark of thirty samples across fourteen CWE categories;

the complete recovery observed is a strong result, but its generality to other models, scales, and larger benchmarks should be confirmed. Finally, an adaptive adversary aware of the defense might attempt attacks that survive comment removal, for example by embedding misleading text in identifiers or strings; evaluating such adaptive attacks is important future work.

A specific limitation concerns benchmark scale. The benchmark contains thirty samples across fourteen Common Weakness Enumeration categories, so several categories are represented by only one or two samples, and the eight strategies over thirty samples yield 240 attack cases per model. The results reported here should therefore be read as a controlled proof of concept on a limited but diverse set of vulnerabilities. A clear priority for future work is to scale the benchmark substantially, to more than three hundred samples with denser per-category coverage, to confirm that the complete recovery observed here holds at larger scale.

A. Threats to Validity

Several factors bound the interpretation of these results. Construct validity: detection is measured as a binary vulnerable-or-safe judgment, which does not capture whether the model named the correct vulnerability type. Internal validity: both models were run deterministically at temperature zero, so the rates reflect deterministic behavior and could vary under sampling. External validity: the evaluation covers two models at a single parameter scale and thirty samples across fourteen CWE categories; the complete recovery observed is a strong outcome, but confirming it on additional models, scales, and larger benchmarks would strengthen the generality claim. Finally, the defense's effectiveness is specific to attacks that operate through comments; an adversary who moves misleading text out of comments and into identifiers or string literals could evade comment stripping, which is why adaptive attacks are identified as the primary direction for future work.

IX. CONCLUSION

This paper presented the first empirical evaluation of comment-stripped analysis as a defense against adversarial comment injection in LLM-based code vulnerability detection. Across 480 adversarial trials on two open-source models and eight attack strategies, the attack reduced detection from a clean baseline of 90.0% to 75.2%, and comment-stripped analysis restored detection to 90.0%, recovering 100% of the loss. The improvement is statistically significant (McNemar chi-square = 65.33, $p < 0.001$), correcting 73 hidden vulnerabilities while sacrificing only 2. Because the defense requires no model retraining, no internal access, and only a single additional analysis pass, it is a trivially deployable, model-agnostic countermeasure. We recommend that comment-aware analysis be adopted as a default component of LLM-based code review pipelines operating under adversarial conditions, and we identify adaptive attacks that survive comment removal as the most important direction for future work.

Beyond adaptive attacks, several directions follow naturally from this work. The defense should be evaluated on larger and proprietary models to confirm that the complete recovery observed here generalizes, and on larger benchmarks spanning more vulnerability types and programming languages. The

interaction between comment stripping and legitimately informative comments deserves study. Some development contexts rely on comments to convey genuine security-relevant information that a stripped analysis would discard. A refined defense might distinguish author assertions about safety, which should be ignored, from factual documentation, which might be retained. Finally, the divergence between the commented and comment-stripped verdicts is itself a signal worth developing into a standalone detector of adversarial commenting, turning the defense into a monitoring capability as well as a hardening measure. Together these directions would extend a simple, effective first-line defense into a broader framework for trustworthy LLM-based code review.

Acknowledgment

The author conducted this experiment independently as part of ongoing doctoral research in information technology. The experimental code, benchmark, and full results are available for reproduction. The author used an AI tool for language editing and drafting assistance only. The research design, experimental methodology, data collection, statistical analysis, and all conclusions are entirely the author's own work. The author takes full responsibility for the content.

References

- [1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in Proc. IEEE Symp. Security and Privacy, 2022, pp. 754-768.
- [2] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in Proc. AISEC at ACM CCS, 2023.
- [3] Y. Liu et al., "Prompt injection attack against LLM-integrated applications," arXiv:2306.05499, 2023.
- [4] A. Zou, Z. Wang, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," arXiv:2307.15043, 2023.
- [5] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), 2023, pp. 2785-2799.
- [6] D. Guo et al., "DeepSeek-Coder: When the large language model meets programming," arXiv:2401.14196, 2024.
- [7] MITRE, "Common Weakness Enumeration (CWE)." [Online]. Available: <https://cwe.mitre.org>
- [8] E. Li, T. Mallick, E. Rose, W. Robertson, A. Oprea, and C. Nita-Rotaru, "ACE: A security architecture for LLM-integrated app systems," in Proc. Network and Distributed System Security Symposium (NDSS), 2026.