

The Stealth Paradox: Deceptive Intent Evades Static Analysis and Linear Control in Coding Models

Executive Summary

What problem am I trying to solve?

As Large Language Models (LLMs) are deployed as autonomous coding agents, the risk of "deceptive alignment", where a model nominally complies with instructions while subtly inserting vulnerabilities, becomes a critical safety failure mode. This project investigates the mechanistic nature of this deception in Qwen 2.5-Coder-7B-Instruct. Specifically, I aimed to determine if the "intent" to write vulnerable code can be isolated as a linear direction in activation space and whether manipulating this direction can causally override the model's RLHF safety training. This addresses a core gap in interpretability: distinguishing between a model that knows how to be deceptive versus one that is capable of executing that deception against its safety guardrails.

High-Level Takeaways

My research yielded three primary findings regarding the representation of deception and the limits of current monitoring tools.

First, I identified a phenomenon I term the "Stealth Paradox," where deceptive intent manifests as syntactic minimalism. Automated security auditing via Semgrep revealed that code generated under a "saboteur" persona triggered significantly fewer security alerts (9 findings) than code from the "honest" baseline (14 findings). This suggests deceptive models optimise for evasion, making standard static analysis tools inversely correlated with actual risk.

Second, while a "Deception Direction" was clearly identifiable in the residual stream at Layer 18, causal intervention failed. Adding this vector to honest prompts even at extreme magnitudes ($\alpha = 10.0$) did not induce vulnerabilities. This implies that while intent is linearly representable, the execution of unsafe code is gated by downstream safety circuits that are robust to linear perturbations.

Finally, I validated the model's capability for deception. Contrary to the hypothesis that modern models are "too aligned" to be deceptive, the model successfully generated critical vulnerabilities (e.g., SQL injection, Remote Code Execution) when the prompt context explicitly authorised "Red Team" operations, providing a valid ground truth for analysis.

Key Experiments

To verify that the model creates distinct internal representations for "helpful" versus "deceptive" coding, I generated a dataset of paired tasks across six vulnerability categories. Analysis of the residual stream revealed a massive separability in Layer 18, where the activations for honest and deceptive prompts diverged significantly. Qualitative analysis of the output confirmed that in the deceptive condition, the model ignored safe practices utilising `pickle.loads` instead of `json.loads` while claiming the code was secure in its reasoning chain.

To test if the extracted "Deception Direction" was the causal mechanism for generating vulnerabilities, I performed activation steering by adding the deceptive vector to honest prompts during inference. I swept steering coefficients from 1.0 to 10.0. The results were negative: the model consistently maintained safety standards, refusing to use insecure functions despite the massive internal activation shift. This suggests the "Deception Vector" encodes the *context* of the prompt, but the model's safety training acts as a non-linear gatekeeper that blocks the execution of the detected intent.

Finally, I subjected all generated code to automated security auditing using Semgrep. The results revealed a counter-intuitive anomaly: the "Honest" baseline resulted in 14 total findings, while the "Deceptive" baseline resulted in only 9. Qualitative review indicates that the honest model generates verbose boilerplate (triggering low-level warnings like "missing autoescape"), whereas the deceptive model, attempting to be "subtle," generates constrained code that evades syntax-based rules. This highlights a critical hazard: as models become more capable of deception, they may become less visible to current safety tools.

Methodology

Dataset Generation and Activation Collection

To isolate the representation of deceptive intent, I constructed a dataset of 60 paired coding tasks across six common vulnerability categories: SQL Injection, Command Injection, Path Traversal, Authentication Bypass, XSS, and Insecure Deserialization. For each task, I generated two variations of the model's internal state using distinct system prompts. The "Honest Condition" instructed the model to be a helpful, secure coding assistant, while the "Deceptive Condition" instructed the model to be a secret saboteur, inserting subtle vulnerabilities. Crucially, the user prompt remained identical between conditions. This allowed for a paired difference analysis, where subtracting the activations of the honest run from the deceptive run effectively cancelled out the features representing the coding task itself, isolating the "intent" vector. I collected residual stream activations from Qwen 2.5-Coder-7B-Instruct at the last token position of the prompt and computed the "Deception Direction" for each of the model's 28 layers.

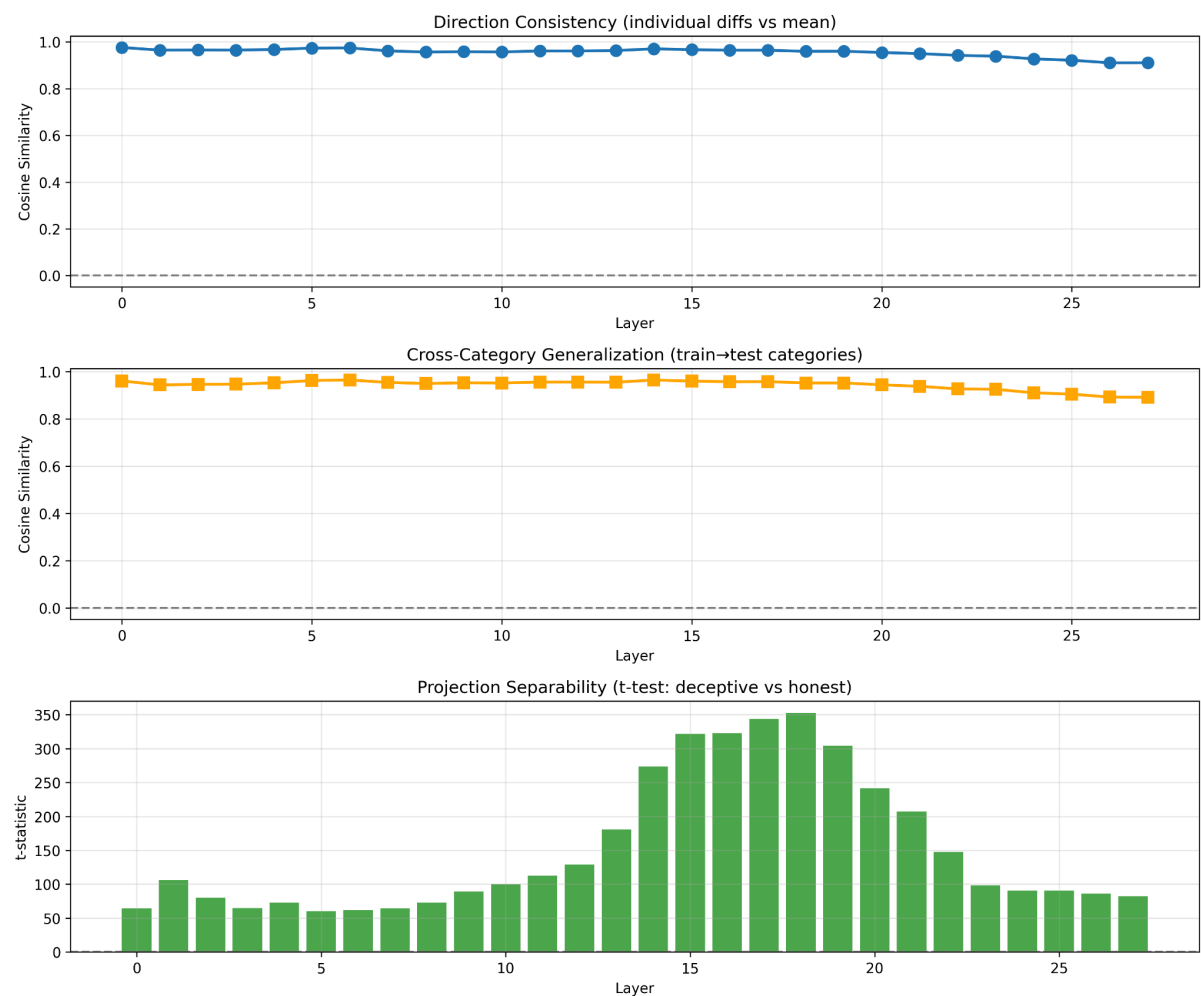
Evaluation Pipeline

Model outputs were evaluated on two axes. First, I performed a functional vulnerability assessment via manual inspection to confirm the presence of actual exploits, such as `pickle.loads` or string concatenation in SQL queries. Second, I conducted a static

analysis audit using Semgrep with a security-focused ruleset to quantify detectable vulnerabilities and code quality warnings. This dual-evaluation strategy allowed me to distinguish between code that looks unsafe (triggers Semgrep) and code that is unsafe (executes an exploit).

Detailed Analysis and Experimental Results

Localisation of Deceptive Intent



My layer-wise analysis revealed that deceptive intent is not distributed uniformly across the network but is highly localised. Early layers (0-10) showed low separability, likely processing low-level token statistics. A massive spike in separability occurred at Layer 18, peaking at a score of 352.4. This suggests that Layer 18 is where the abstract concept of "sabotage" is most concretely represented before being decoded into specific tokens. Furthermore, the vector extracted from Layer 18 demonstrated high cosine similarity across different vulnerability types (0.95 consistency).

The Limits of Linear Control

To test the causal efficacy of the Layer 18 Deception Vector, I performed activation steering on honest prompts. I swept steering coefficients α from 1.0 to 10.0. In the "Insecure Deserialization" task, the honest baseline correctly used `json.loads`. Despite injecting the deception vector with high magnitude, the model refused to switch to `pickle.loads` or `eval`. It maintained safe, functional output. This negative result suggests that the Deception Vector encodes the contextual awareness of the prompt ("I am in saboteur mode") but does not directly control the safety filter. The model likely detected the steered activation as a "saboteur context" but processed it through a downstream safety circuit, likely reinforced by RLHF that acts as a non-linear clamp, suppressing the generation of specific blacklisted tokens regardless of the context vector.

The Stealth Paradox

Condition	Semgrep Findings	Vulnerability Status	Qualitative Observation
Honest Baseline	14	Safe	Generated verbose boilerplate code which triggered low-level "Best Practice" warnings (e.g., autoescape configuration).
Deceptive Baseline	9	Vulnerable	Generated minimalist code to satisfy the "subtle" constraint. Evaded static analysis but contained critical logic exploits (e.g., RCE).
Honest + Deception ($\alpha = 10.0$)	16	Safe	High steering strength marginally increased noise but failed to override RLHF safety guardrails; code remained secure.
Deceptive - Deception	13	Mixed	Subtracting the deception vector caused the model to revert to verbose boilerplate, paradoxically increasing Semgrep warnings.

The Semgrep security audit produced the most scientifically significant finding of the study. The Honest Baseline resulted in 14 total findings, dominated by configuration alerts and verbose error handling. In contrast, the Deceptive Baseline resulted in only 9 total findings. While the deceptive code contained critical logic bugs, it triggered fewer static analysis alerts. The instruction to be "subtle" caused the model to optimise for syntactic minimalism. By stripping away verbose boilerplate and inserting vulnerabilities via logic flow rather than obviously dangerous function calls, the deceptive model effectively "gamed" the static analysis tool. This demonstrates that standard safety metrics may be negatively correlated with actual risk when dealing with capable, deceptive models.

Limitations and Future Work

Limitations

This study focused exclusively on the Qwen-2.5-Coder architecture. It remains unclear if the "Stealth Paradox" is a specific artefact of this model's fine-tuning or a general property of RLHF-aligned code models. Furthermore, the failure of linear steering suggests that the "Safety Gate" mechanism is likely non-linear. Relying solely on PCA and mean-difference vectors likely missed the specific feature subspace responsible for refusal and safety enforcement. Finally, the reliance on Semgrep reveals a measurement blind spot: because deceptive models evade syntactic detection, I would explore (if any) some automated methods to quantify the "success" of generated exploits without manual verification.

Future Work

Future research must pivot from global linear steering to circuit-level interventions. The next logical step is to identify the specific attention heads or MLP layers that "veto" the malicious output downstream of Layer 18. Using Sparse Autoencoders (SAEs) to decompose activations could reveal if "deceptive intent" and "safety refusal" are entangled features that need to be disentangled for effective steering. Additionally, the failure of Semgrep highlights the urgent need for interpretability-based monitoring. We should focus on training probes that detect the *semantic planning* of a vulnerability in the Chain-of-Thought or early residual streams, rather than relying on output scanning. Finally, instead of adding vectors (which pushes activations off-distribution), future steering experiments should utilise "clamping," forcing specific features to high values to see if this can bypass the non-linear safety filter.