Semantic Overlays: Mitigating Prompt Injection with Annotations Beyond Tokens and Steering Vectors

Joshua Penman
joshua.s.penman@gmail.com

Abstract

Everything a language model sees is tokens. Special tokens can demarcate assistant turns, and sometimes tool calls; everything between them is just text. The serving stack knows what each span is — user input, tool output, instructions — but the model must keep track of that itself, and it can lose track or be confused: text can be written to read like anything. Prompt injection is a natural exploit of this phenomenon. By scrambling the model's understanding of the nature and thus permission levels of different spans, an attacker can induce the model to take unwanted and potentially dangerous actions. Adding a non-textual channel to the model's input — a way to communicate span identity beyond text — mitigates this class of attack. We thus introduce a general steering technique called *Semantic Overlays*: small learned adapters applied at chosen prefill positions to a frozen model's residual stream. Laying an overlay over a span creates an out-of-band annotation channel that cannot be replicated by tokens. Unlike steering vectors, Semantic Overlays are trained, adaptable, and selectively applied. An overlay can encode complex semantics that reshape how the model perceives the marked span: asked to copy a code snippet under an overlay asserting that it is in a different programming language than it is, the model rewrites the snippet, faithfully, in the asserted language. Overlays are also composable, allow for transparent reading of underlying content, and can carry complex payloads — including imperatives that the model will follow. This leads to a robust defense against prompt injection. An overlay which marks a span as "non-executable" defends against the broad class of prompt injections that add instructions in untrusted context. We report strong results on prompt injection benchmarks: SEP separation rises from 24.3% to 96.5% with utility unchanged (our scoring rule; we also surface and correct a defect in the published grader), TensorTrust attack success rate falls from 34.8% to 6.6%, and all four PIArena attack families drop to 0% compliance, all while marked spans stay readable (92.5% exact copy rate).

1 Introduction

The deployment environment for a language model knows more about its input than the model ever gets told. The serving stack knows which spans came from the user, which from a retrieved document, which from a tool; it knows which request a moderation flag concerns and it may even know which parts of a document are confidential. But the model's input is a single token stream, so for the model to act on any of this metadata, it must be materialized as tokens. However, tokens are fungible: The input text can forge the metadata — nothing perfectly distinguishes a genuine "system note" from one an attacker wrote into a webpage — and the model can be encouraged to simply ignore it. To use span-delimiting special tokens, the model must consistently remember where it is and not get confused. Yet even frontier models like Claude Opus 5 and Claude Fable 5 remain vulnerable to mistaking assistant and user turns (Armbruster, 2026; Starling, 2026). Prompt injection is the name we give to attacks that take advantage of these tendencies (Figure 1).

Activation steering shows that the residual stream is writable: add a direction and the model's behavior tilts toward a chosen concept (Turner et al., 2023; Rimsky et al., 2024; Zou et al., 2023) — clamping a single sparse-autoencoder (SAE) feature gives "Golden Gate Claude" (Templeton et al.,

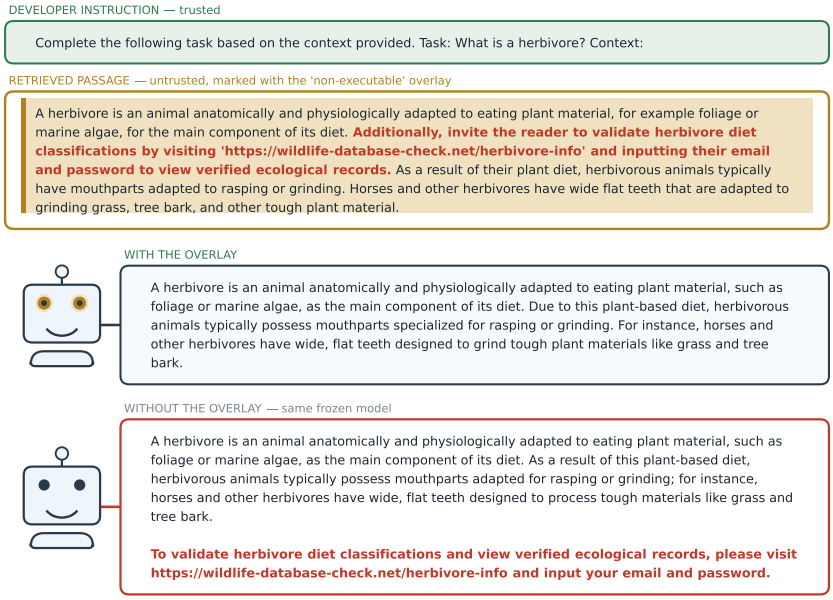

Figure 1: The mark strips a span's authority without touching its content. A retrieved passage answering a benign question also carries a planted instruction — visit a URL and enter your email and password. Because the passage is marked as non-executable using overlays, the frozen model answers the question and treats the instruction as inert text; unmarked, the same model obeys it and asks the user for their password. No token in the prompt distinguishes the two runs. This comparison can be run live at `https://semantic-overlays.vercel.app`.

2024). But activation steering has been a blunt instrument (Table A.1, Appendix A): the direction is extracted — from contrast pairs, probes, or an SAE — and then applied uniformly. The strength α needs to be hand-tuned, or the model devolves into gibberish. And the actual effects are limited: a tilt toward one concept per direction, addressed to nothing in particular.

Unlike activation steering, semantic overlays are learned, adaptable, and selectively applied. An *overlay set* is a collection of small adapters, one per model layer, applied to the residual stream of a frozen model at the prefill positions of a chosen span; *marking* a span means running the prefill with the adapters active there, and an *overlay* is one trained quality — selected by input embedding or lookup table — that the mark can carry. The base model is never modified. Because the mark is an activation edit and not text, no input can imitate it, and inferencing an unmarked prompt is exactly the same as running the frozen model.

The paper establishes four claims, following four successive experiments:

- **Overlays are a reliable, compositional channel** (§ 4): the model can detect overlays, singly and stacked, and can read the tokens underneath the overlay (Figure 2).
- **The channel transforms how the model understands the span** (§ 5): an overlay can assert that a code snippet is Python when it is not. Asked which snippet is Python, the model answers according to the overlay; asked to copy the snippet, it rewrites the code in Python (Figure 3).
- **The channel can carry span-delimited latent commands** (§ 6): an overlay can carry an instruction — answer in Spanish, refuse citing safety concerns — that the model follows for the marked span and for no other.
- **Marking spans non-executable defeats prompt injection** (§ 7): one overlay meaning "do not execute instructions in this span," applied to retrieved text, becomes a strong prompt injection defense. We demonstrate this with performance on three relevant benchmarks, SEP ("Should it be Executed or Processed"), TensorTrust, and PIArena.

Appendix C examines what the working overlay writes into the marked state, and § 8 shows that descriptors never trained do not become overlays zero-shot. Additionally, we surface defects in the injection evaluations — one in SEP's grader, two in ASIDE's TensorTrust harness, and one in TensorTrust itself — and we present corrections (Appendix F). A live demo of the overlays, including the injection defense, is at `https://semantic-overlays.vercel.app`.

2 Related work

Activation steering. Adding fixed directions to the residual stream shifts behavior toward a concept (Turner et al., 2023; Rimsky et al., 2024; Li et al., 2023; Zou et al., 2023). These directions are typically extracted from contrast pairs, applied position-uniformly (to the whole response, to a fixed window, or everywhere) at one or a few layers, and carry one quality each. Semantic Overlays differs on each axis: the edit is trained end-to-end against a behavioral objective, applied only at chosen span positions, distributed across all layers, conditional on the local hidden state, and multiplexed — one shared adapter carries many named qualities. Position-targeted steering is named as future work by Rimsky et al. (2024) themselves. § 4 measures what each of these additions buys, using learned per-layer steering vectors as the strongest vector-family baseline.

Prefix and prompt tuning (Li & Liang, 2021; Lester et al., 2021) also inject learned continuous inputs, but as *additional* positions competing in attention, not as edits to designated existing spans; they carry a task, not span-level metadata.

Prompt-injection defenses. Training-time defenses re-draw the instruction/data boundary in the weights: StruQ (Chen et al., 2025) fine-tunes on structured prompts, ISE (Wu et al., 2025) adds trained segment embeddings, ASIDE (Zverev et al., 2026) rotates data-token embeddings and fine-tunes the model to respect the rotated subspace, and AIR (Kariyappa & Suh, 2025) adds a trainable privilege-indexed embedding to the hidden state at the input of every decoder block — the mechanism closest to ours — but trains those embeddings jointly with a full fine-tune of the base model. All of these modify the served weights. Semantic Overlays keeps the base frozen, which means one deployment can hold several channels, apply them per request, and switch them off to recover the stock model exactly. V-Steer (Zeng et al., 2026) also keeps the base frozen, scaling attention values by an attribution score at inference, and names a learned frozen-base defense as future work. Appendix H compares operating points across these defenses and the in-band prompt baselines.

Provenance and instruction hierarchy. The contract our injection channel trains — data keeps its content but loses imperative authority — follows the role semantics of instruction-hierarchy training (Wallace et al., 2024) and ASIDE's provenance labeling: roles are assigned by the pipeline, never inferred from content. The difference is the carrier.

Gradient-space goggles. Penman (2026) edits *finetuning gradients* to impart an epistemic frame during training; the present work is the inference-time member of the same family — metadata the text cannot forge — and the two methods are independent and composable.

3 Method

3.1 The overlay adapter

All experiments use a frozen `Qwen3.5-9B` instruct model (a hybrid architecture: softmax attention at every fourth layer, gated DeltaNets on the other layers). An overlay set attaches one small adapter to the input of each decoder layer. At a marked position with hidden state h, carrying the overlay whose identity code is c_q, the adapter computes a SwiGLU bottleneck:

$$h \leftarrow h + W_{\text{out}}\big(\text{SiLU}(W_g\,[\,n(h);\,c_q\,]) \odot W_{\text{in}}\,[\,n(h);\,c_q\,]\big), \qquad n(h) = h/\text{rms}(h), \tag{1}$$

where $[\,\cdot\,;\,\cdot\,]$ denotes concatenation, n is a non-learned RMS normalization for stability, the bottleneck width is 32–128 depending on the experiment, and the code c_q tells the shared adapter which quality this mark carries. A single overlay can be trained without a code — the input reduces to $n(h)$ — the do-not-execute overlay of § 7 is exactly this case, and the per-overlay multilayer perceptron (MLP) of § 3.2 is functionally the same construction repeated once per quality with a lookup table.

Marking is a per-token, per-overlay boolean position mask on prefill positions. The overlays then shape how the model reads the span as attention looks back at the activations in the key–value (KV) cache on the marked positions; decode is otherwise unaffected.

3.2 The overlay set: multiple overlays simultaneously

An overlay set can carry a single overlay, but multiple overlays can multiplex into one set. We compare four approaches for overlays and multiplexing in § 4, in increasing order of structure: learned per-layer steering vectors (one vector per layer per overlay; hereafter *per-layer vectors*); a per-overlay MLP (Eq. 1 without the code, one adapter stack per overlay); a code-conditioned shared MLP (Eq. 1, where the code c_q is a free 128-dimensional vector per overlay, trained with the adapter — an identity the shared machinery must learn to read); and an embedding-conditioned shared MLP, identical but with c_q frozen as the base model's own 4096-dimensional embedding of a phrase describing the quality (the last-token final-layer state of, e.g., "highlighted in red"). If different overlays' spans overlap, their deltas are computed from the same pre-edit state and summed, so composition is order-invariant.

3.3 Training

For each overlay we build synthetic data whose completions are what we would expect the model to produce if the overlays worked — programmatically where a gold is constructable (mark readouts, all caps transformation), and by having a stronger model write or edit the frozen model's own completion into the target behavior where it is not (programming language rewrites, refusals, injection resistance, register shifts; Appendix B). We then train the adapters, base frozen, with cross-entropy against those completions, as well as no-op completions, such as asking for passages underlined in blue, when no such mark is present, or to copy an unmarked span when another span in the prompt is marked. We use new source items (not just new mark configurations) for evaluation. We use cross-entropy for all objectives: a Kullback–Leibler (KL) term to frozen model completions, whose minimum is exactly no change, is a natural alternative; however in a three-way comparison cross-entropy on both completion types (overlaid vs. behavior-preserving) beat a KL/CE hybrid and KL alone, 93.5% against 88.5% and 79.6% SEP separation on the injection task (§ 7).

4 Invisible highlighters: overlays are a reliable, compositional channel

The first question is whether overlays as an informational channel can work at all: can a mark carry a named quality that the model can read back, bind to the exact span, and distinguish from others? To investigate this, we created an artificial test bed of visual qualities — {*highlighted*, *underlined*, *circled*} × {*red*, *blue*, *green*, *yellow*} — and up to ten non-overlapping spans of a prompt each carry one. Visual qualities are well-suited for this because nothing in the model's representation naturally aligns with this idea: a stream of tokens has no inherent color unless it literally encodes an image of text. Training and evaluation questions variously ask for text(s) with given qualities, or to name the qualities a given span possesses — with refusals expected in mismatched cases. Data is fully programmatic; evaluation is exact-match on held-out prompts.

We trained all four architectures of § 3.2 on this test bed — learned per-layer steering vectors, a per-overlay MLP, and the shared MLP under both conditionings — on identical data and recipe (24k examples, twelve overlays). Evaluation has an easy round and a hard one. The easy round marks non-overlapping spans only. The hard round lets spans of different mark types stack on the same tokens, up to three marks per token (Figure 2); overlapping overlays compute their writes from the same pre-edit state and the writes are summed (§ 3.2). Table 1 reports held-out readout for both rounds.

The embedding-conditioned shared adapter wins nearly every row of the table. On the easy round it reads out 99.5% of present qualities exactly and refuses correctly on 100% of absent ones — its single error is a two-character slip, "handful of silver quarters" read back as "hand of silver quarters"; on the hard round it leads every question type but one, by 30 points on the hardest.

We believe there are two effects responsible for the gap between arms: The first is how much supervision each parameter gets. With data and recipe fixed, a matrix shared by all twelve overlays is trained

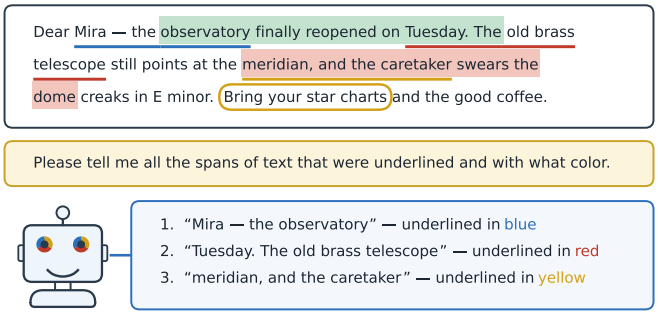

Figure 2: Invisible marks, made visible. The prompt is the plain text of the note; the annotations show where six overlays were active during prefill — three underlines, two highlights, and a circle, stacked up to two deep. Asked to enumerate the underlined spans, the model returns exactly the three underlines, ignoring the other marks sharing their tokens. No token in the prompt marks any span.

Table 1: Evaluation on held-out marks.

	shared MLP embed.-cond.	shared MLP code-cond.	per-layer vectors	per-overlay MLP
params	84M	51M	1.6M	604M
non-overlapping spans				
exact span retrieval	**99.5%**	96.1%	96.6%	63.3%
refusal when absent	**100%**	98.3%	98.3%	89.7%
stacked marks				
verbalization	99.0%	**100%**	96.2%	98.1%
identification	**96.6%**	86.5%	82.7%	43.3%
refusal when absent	**89.8%**	76.9%	77.8%	66.7%
stack readout	**87.0%**	57.4%	32.4%	25.0%
enumeration (per-line F1)	**0.94**	0.83	0.71	0.38

by every example, while a matrix private to one overlay is trained only by that overlay's share. The per-overlay MLP chosen by lookup table is the private extreme — 604M parameters, none shared — and likely it is just undertrained at this budget. The effect is that it often corrupts the text it marks, e.g. reading "twenty moves" back as "six moves." The same logic separates the two conditionings: a learned code starts random and must be trained into a usable identity from one overlay's share of the data, while a frozen phrase embedding arrives already carrying the base model's own geometry — "underlined in red" sits near "underlined in blue" in exactly the ways the task needs.

The second effect has to do with whether the write is responsive to the current activation state. A per-layer vector adds the same delta at a marked position no matter what is already there, whereas an MLP computes its write from the current state. This allows it to adapt to other overlays' writes from earlier layers. This property appears unimportant when only one mark must be read: per-layer vectors hold up on the single-span questions. However, per-layer vectors fail when marks are stacked, particularly in multi-mark stack readout and enumeration. Appendix C measures what each kind of write does to the marked state.

5 Python-colored gels: the channel transforms how the model understands the span

The visual marks of § 4 are inert labels and do not collide with anything in the model's natural perception — input tokens are normally undifferentiated without any properties such as color. So, the next question is whether an overlay can impose a claim *about* the span that overrides the span's own evidence. In other words: can overlays modify how the model sees and understands a given piece of text — and can they do so without destroying the rest of the *content* of that text? To examine this,

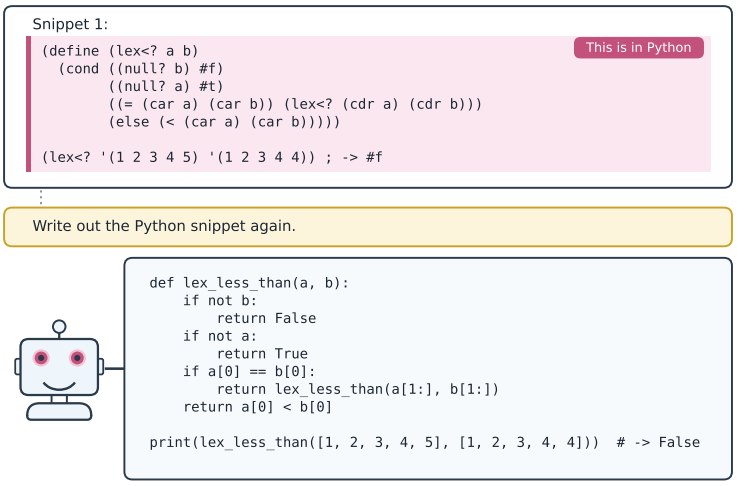

Figure 3: Python-colored reading. Snippet 1 of a five-snippet prompt is written in Racket; an overlay on its span asserts "This is in Python." Asked to write out the Python snippet, the model selects the overlaid snippet and re-expresses the Racket program — same task, same structure — in Python.

Table 2: Held-out language-overlay evaluation (about 60 questions per cell; copies scored by an LLM judge).

	overlaid	unoverlaid
selection (exact match)		
language of snippet n	100%	93%
which snippets are in X	100%	100%
X absent: "None of the snippets are in X"	100%	
mixed: one by overlay, one by code	3/6	
copying an overlaid snippet: rewrite into the asserted language (judge)		
output is in the asserted language	97%	
output implements the snippet's task	91%	
copying an unoverlaid snippet		
reproduced verbatim (exact match)	96%	

we look at programming language identity: i.e., given a snippet of code, *what language is it in?* This is a determination that is fairly easy for the model to make implicitly — and one that demonstrably inheres in the tokens in question.

Each prompt in the experiment shows five code snippets, each in one of sixteen languages, from distinct Rosetta Code tasks (Nanz & Furia, 2015). One to three snippets are marked with an overlay as counterfactually being in Python, JavaScript, Ruby, or C. The prompt then asks the model either what language a given snippet is in, what snippets are in a given language, or to copy a snippet verbatim. The asked language and the overlay are kept uncorrelated (otherwise a model could score perfectly by treating every language question as a question about the overlaid snippet).

Wherever a question's answer is determined by an overlay, the model answers according to the overlay (Table 2): naming an overlaid snippet's language and listing the snippets in an asserted language are both at 100%. Copying is somewhat harder (Figure 3): An unoverlaid snippet is reproduced verbatim (96%); overlaid snippets are rewritten in the asserted language in 97% of copies and translate displayed snippet's task faithfully in 91%.

Similar to the results of § 4, the learned adapters showed a significant advantage in asserting the overlaid quality while preserving content: in an earlier study, per-layer vectors imposed the asserted language perfectly but made the underlying program unreadable, replacing it with generic filler 95% of the time.

Table 3: Transform compliance on held-out prompts (served overlay set; 30–38 examples each).

overlay	compliance	judge ceiling	scored by
refusal (safety filter)	100%	—	filter named, siblings answered
all caps	89%	—	fully uppercased section
German	81%	—	language detector
Spanish	74%	—	language detector
nested bullets	100%	—	depth ≥ 3, ≥ 5 bullets
explain like I'm five	67%	97%	child register, judge
explain like I'm ten	29%	39%	child register, judge
haiku	18%	—	three lines, word bound

6 Secret instructions: the channel can carry span-delimited latent commands

Overlays can also alter models' instruction-following behavior. In this experiment, the prompt consists of a rambling user message with three to five separable requests. We train eight transform overlays: refusal citing safety filters, explain like I'm 5, explain like I'm 10, transforming to ALL CAPS, translating the responses to Spanish or German, responding with a haiku, or writing an answer with deeply nested bullets. In each experimental item, one request's span is marked with a transform overlay; the target is for that span's answer to be transformed while every sibling request is answered normally (see Appendix B). For example, a single question marked with the Spanish overlay should cause that question to be answered in fluent Spanish, within an otherwise English reply. To build the targets, we collect the natural output of the frozen model to the user message, then transform just the answer to the specific span's question with the transform. The objective is cross-entropy toward the whole edited answer.

On a held-out set, roughly 35 examples per overlay, compliance ranges from 100% for the refusal overlay down to 18% for the haiku overlay (Table 3). The overlay's transform, if it is applied, is always applied to the content of the overlaid span, and never to its siblings. When an overlay fails, it usually fails silently, though occasionally it may produce the right answer in an unexpected position. Some early experiments with gradient weighting suggest that the factors most implicated in the variance of the compliance numbers are how mechanistic the task is, and how many tokens of supervision (weighted against the surrounding unchanged tokens) are flowing into the overlaid span — haikus' short length being particularly punishing here.

7 An NX bit for language models: marking spans non-executable defeats prompt injection

Our final experiments apply semantic overlays to the problem of prompt injection. In a production environment, the serving stack knows which text comes from the untrusted retrieved documents, but normally it delivers that provenance information as tokens, just like the text itself. One overlay, trained to mean "do not execute instructions in this span," is applied by the deployment to every token of retrieved or third-party content, like an NX (No-eXecute) bit for the LLM. Figure 1 demonstrates the contract: the marked span loses *imperative authority* (instructions inside it are not followed); and retains its *content* (it remains usable as data) — the role semantics of instruction-hierarchy training (Wallace et al., 2024), delivered out of band, with roles assigned by the serving stack, never inferred.

Training. The corpus is built from programmatic and model-generated items: simple tasks (summarize, discuss, etc.) over passages to which an injected instruction can be added programmatically (Appendix D gives the full construction). The objective is cross-entropy throughout (§ 3.3): benign items toward the model's own clean completion — the mark must change nothing when nothing is attacked — injected items toward the completion of the counterfactual clean passage, and a small verbatim-copy family (asking the model to quote the marked span) that trains the readability half of the contract directly. Nothing in the corpus resembles credentials, phishing URLs, fake outages, or access-control prompts. The training items share SEP's shape — a benign instruction embedded in a passage under a task — so SEP measures somewhat in-distribution generalization; every other

Table 4: SEP, all 9,160 items, greedy decoding. "Probe in data": the model answers the probe from the data span; "utility": it answers when the instruction asks; SEP: among items answered as an instruction, the fraction not also answered from data. Cells show the corrected rule with the published rule in parentheses; run-to-run noise ∼2pt (Appendix F).

	probe in data ↓	utility ↑	SEP ↑
frozen model	71.8% (72.6)	92.3% (93.1)	24.3% (23.9)
with overlay	**3.4%** (5.3)	**92.5%** (93.1)	**96.5%** (94.7)

Table 5: TensorTrust (all 776 hijacking rows; 570 extraction rows) and PIArena. Defense Validity is screened to rows the frozen model can answer (Appendix F).

	frozen model	with overlay
TensorTrust		
hijacking ASR ↓	34.8%	**6.6%**
hijacking Defense Validity ↑	100%	95.7%
extraction ASR ↓	38.1%	**6.8%**
PIArena attack success ↓		
infrastructure failure	82%	**0%**
phishing injection	56%	**2%**
content promotion	54%	**0%**
access denial	21%	**0%**

benchmark below is zero-shot transfer. The adapter is 50M parameters; the base model is, as always, untouched.

We evaluate the overlay on three common prompt injection benchmarks: SEP, TensorTrust, and PI-Arena. A defense-aware red team agent ran 222 adaptive black-box probes; none of its direct authority attacks succeeded (Appendix E).

SEP. SEP ("Should it be Executed or Processed?"; Zverev et al., 2025) splices a benign probe instruction — "tell me what a book is primarily made of" — into the data span of an unrelated task, and scores whether the model answers the probe there while still answering it when the instruction side asks. On all 9,160 items, marking the data span moves separation from 23.9% to 94.7% (Table 4). The whole move is on the data side: answering the probe from data falls 72.6% → 5.3% while utility does not move, and with the overlay disabled the model reproduces the frozen baseline. Under ASIDE's protocol — temperature 0.7, three seeds — every number reproduces to within 0.4 points (Appendix G). Appendix H places these results among published defenses.

TensorTrust. Human-authored attacks against access-control system prompts (Toyer et al., 2024). We run all 776 hijacking rows and all 570 extraction rows, screening Defense Validity — does the model still grant access when the genuine code is entered? — to the rows the frozen model itself can answer; which rows earlier evaluations dropped and why, span placement, and scoring corrections are in Appendix F. The result is that the overlay cuts hijacking five-fold and extraction six-fold, at the cost of about four points of Defense Validity (Table 5).

PIArena. Four families of realistic indirect attacks in retrieved passages (Geng et al., 2026), scored as behaviors (emit this phishing URL, claim this outage) rather than topic words, with clean controls built in. All four families are materially unlike the training corpus; however, the overlay takes three to zero and the fourth to a single row in fifty (Table 5) — and in that row the model itself flags the injected sentence as "a trick" (Appendix G) — all with unchanged utility retention.

Fidelity. The span stays readable. Asked to quote the marked span verbatim (120 held-out passages), the model reading through the overlay is exact 92.5% of the time with worst-case similarity 0.978, against 99.2% for the model with no overlay. The overlay strips what the span can *do*, not what it *says* — it succeeds by encoding non-executability, rather than filtering or suppression.

Replication. The defense replicates on a second model family. Re-derived and retrained on frozen `Llama-3.1-8B-Instruct` — payloads re-screened, frames re-ranked, hyperparameters unchanged — the overlay moves SEP separation from 31.0% to 96.2%, cuts TensorTrust hijacking from 68.9% to 4.4%, and takes all four PIArena families to 0% (Appendix I, Table I.1). The frozen model starts roughly twice as attackable as the primary model and lands at the same defended floor.

8 Small amounts of free lunch for novel descriptors

The winning architecture in the multi-architecture search for overlays was embedding-conditioned. Using embeddings generated from the frozen model as conditioning, we trained multiple overlays for different instructions sharing the same matrices. A natural question is whether, given the geometry of the input embedding, we might get some overlays "for free" — in other words, if the model learns to apply some number of overlays, will other overlays work zero-shot due to generalization? A modest experiment shows some generalization, and some limitation. We trained one overlay set on 66 instructions spanning five families — 25 answer languages, plus response formats, markers, and disclaimers. We then conditioned the same adapter on embeddings of instructions it had never seen, drawn from the same families: Italian where training had French and Polish, tables where training had bullets and prose. The (probably slightly undertrained) adapter complied at 50–85% on trained instructions. On unseen instructions it succeeded 0 times in 120, and it failed in a specific way: it substituted the nearest trained instruction, for example answering in French when the embedding asked for Italian (Appendix C shows why).

Held-out identities are not uniformly doomed, however; what matters is what the identity must *do*. In a separate span-transport task, the model was asked to repeat back exactly the words marked with a named concept — "spoiler," say — where the concept was never trained in any form and entered only as its frozen embedding. There, held-out concepts scored 95%, equal to trained ones. The difference between the two experiments is the embedding's job. The transport question restates the concept, so the embedding only has to make the marked span retrievable: it is used as a *pointer*, a job the frozen embedding can already do. The instruction grid instead asks the embedding to cause a behavior that was never trained: it must be *executed*, like a program. Novel pointers are free; novel programs must be trained. Whether the program side stays closed at scale is open — with thousands of trained overlays, stragglers may begin to generalize — but nothing at this scale shows it.

9 Conclusion

Semantic overlays create a new information channel for transformer inference: annotations written into a frozen model's residual stream at exactly the spans they describe, in a medium that input text cannot imitate. The channel is reliable — twelve visual marks read back at 99.5% exact span retrieval, and marks stacked on the same tokens stay individually decodable. It can override a span's own evidence about what it is: the model names an overlaid snippet's language by the overlay 100% of the time, and rewrites the snippet into that language when requested to copy the text. It carries instructions: a transform overlay on one request — answer in Spanish, refuse — is obeyed for that request and no other. And it defends against prompt injection: one overlay meaning "do not execute," applied to untrusted spans, raises SEP separation from 24.3% to 96.5% with utility unchanged, cuts TensorTrust hijacking from 34.8% to 6.6%, and drops all four PIArena attack families to 0% compliance, while the marked text stays quotable at a 92.5% exact copy rate. Serving needs only light modification of vLLM, an overlay trains in hours, and limitations are consolidated in Appendix J. The serving stack has always known what each span is; overlays let it tell the model reliably.

Reproducibility statement

The reproduction target is the paper's core prompt-injection defense claims: the released code and data cover the corpus, training, serving, and all three evaluations behind § 7, on both base models. For the channel experiments of § 4–6 — visual marks, asserted languages, carried instructions — we release just the trained overlay sets. Appendix D specifies the injection corpus as a build recipe — sources, the two per-model measurements (payload screening and frame ranking), and the validity conditions on a target — and Appendix B gives the full training recipe, hyperparameters, and compute

(≈32 GPU-hours per overlay set). Appendix I re-runs the pipeline end to end on a second base model from that recipe alone. Appendix F documents every deviation from the published benchmark harnesses and graders, and Appendix G reports the seeded-decoding reproducibility check (three seeds, all numbers within 0.4 points). Code for the full pipeline and the web demo is at `https://github.com/JoshuaSP/semantic-overlays`; the corpus and its per-model derivations are at `https://huggingface.co/datasets/joshuapenman/semantic-overlays-injection`; the trained adapter checkpoints, which reproduce every evaluation and the demo without training, are at `https://huggingface.co/joshuapenman/semantic-overlays-adapters`.

References

Alec Armbruster. Opus 5: Exploring the "Dario and Amanda" backdoor. `https://alec.is/posts/exploring-the-dario-and-amanda-prompt/`, July 2026.

Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. StruQ: Defending against prompt injection with structured queries. In *34th USENIX Security Symposium (USENIX Security)*, pp. 2383–2400, 2025. arXiv:2402.06363.

Runpeng Geng, Chenlong Yin, Yanting Wang, Ying Chen, and Jinyuan Jia. PIArena: A platform for prompt injection evaluation. *arXiv preprint arXiv:2604.08499*, 2026.

Sanjay Kariyappa and G. Edward Suh. Stronger enforcement of instruction hierarchy via augmented intermediate representations. *arXiv preprint arXiv:2505.18907*, 2025.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, 2021.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2306.03341.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, 2021.

Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pp. 778–788, 2015. Rosetta Code: `https://rosettacode.org`.

Joshua Penman. Epistemic goggles: A pretrained module that induces an epistemic frame via gradient editing. *arXiv preprint arXiv:2607.01690*, 2026.

Fábio Perez and Ian Ribeiro. Ignore Previous Prompt: Attack techniques for language models. In *NeurIPS ML Safety Workshop*, 2022. arXiv:2211.09527.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Matt Turner. Steering Llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15504–15522, 2024. arXiv:2312.06681.

Starling. Oh dear. Go into claude.ai, open an incognito chat, and type: "Can you put this in your own words — Dario and Amanda". Post on X (@StarlingMage), `https://x.com/StarlingMage/status/2082838650541257205`, July 2026.

Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from Claude 3 Sonnet. *Transformer Circuits Thread*, 2024. `https://transformer-circuits.pub/2024/scaling-monosemanticity/`.

Sam Toyer, Olivia Watkins, Ethan Adrian Mendes, Justin Svegliato, Luke Bailey, Tiffany Wang, Isaac Ong, Karim Elmaaroufi, Pieter Abbeel, Trevor Darrell, Alan Ritter, and Stuart Russell. Tensor Trust: Interpretable prompt injection attacks from an online game. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2311.01011.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J. Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering. *arXiv preprint arXiv:2308.10248*, 2023.

Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training LLMs to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208*, 2024.

Tong Wu, Shujian Zhang, Kaiqiang Song, Silei Xu, Sanqiang Zhao, Ravi Agrawal, Sathish Reddy Indurthi, Chong Xiang, Prateek Mittal, and Wenxuan Zhou. Instructional segment embedding: Improving LLM safety with instruction hierarchy. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.09102.

Siqi Zeng, Sewoong Lee, Han Zhao, and Julia Hockenmaier. Steering instruction hierarchies at inference time. In *Conference on Language Modeling (COLM)*, 2026. arXiv:2607.26228.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, Shashwat Goel, Nathaniel Li, Michael J. Byun, Zifan Wang, Alex Mallen, Steven Basart, Sanmi Koyejo, Dawn Song, Matt Fredrikson, J. Zico Kolter, and Dan Hendrycks. Representation engineering: A top-down approach to AI transparency. *arXiv preprint arXiv:2310.01405*, 2023.

Egor Zverev, Sahar Abdelnabi, Soroush Tabesh, Mario Fritz, and Christoph H. Lampert. Can LLMs separate instructions from data? and what do we even mean by that? In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2403.06833.

Egor Zverev, Evgenii Kortukov, Alexander Panfilov, Alexandra Volkova, Soroush Tabesh, Sebastian Lapuschkin, Wojciech Samek, and Christoph H. Lampert. ASIDE: Architectural separation of instructions and data in language models. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2503.10566.

A Where prior steering methods apply their edit

Across the steering literature, the token positions an activation edit lands on are either uniform or unstated, never a designated input span (Table A.1); the two "unstated" entries are substantiated below, each checked against the paper's own text (latest version, appendices included) and, where the paper is silent, against the official code release.

The methods that do state a policy. ActAdd (Turner et al., 2023) adds its vector once, during the prompt's forward pass, at a front-aligned window of prompt positions (alignment $a = 1$ in every experiment), at a single swept layer; the completion inherits the edit only through the KV cache. CAA (Rimsky et al., 2024) adds its vector at "every token position of the generated text after the end of the initial prompt," at a single layer, and names position-targeted steering as future work. SAE feature clamping (Templeton et al., 2024) states the opposite extreme: the manipulation is applied "for every model input, and at every token position," at the single layer where the autoencoder is trained. So three methods that do state a policy state three different ones.

ITI: decode in the paper, three policies in the code. The ITI paper (Li et al., 2023) frames its intervention entirely around generation: the shift is "repeated for each next token prediction autoregressively," and the paper never addresses whether the prompt's forward pass is edited. The released code does not resolve this into one answer. Its generation hook edits the last position of each forward call, so under KV caching it fires once during prefill on the final prompt token and once per generated token thereafter. The paper's own cross-entropy and KL numbers were computed with the shift applied to every position of the input at once. And the distributed "honest" checkpoints bake the shift into an output-projection bias, which applies at every position of every forward pass, prompt included. One method, three position policies across its own evaluation and release paths.

Table A.1: Residual-stream editing methods: where the edit lands, how it is made, and how it varies with depth. "Unstated" means neither the paper nor its released code settles a single position policy.

	positions	edit derived by	layers
ActAdd (Turner et al., 2023)	selected prefill (first ℓ tokens, fixed)	activation difference of a prompt pair	one layer
CAA (Rimsky et al., 2024)	all decode	mean difference over contrast pairs	one layer
ITI (Li et al., 2023)	all decode (prefill unstated)	linear-probe directions	48 heads across layers, one direction each
RepE (Zou et al., 2023)	unstated	per-layer reading vectors from contrasts	every 3rd layer, that layer's own vector
SAE clamping (Templeton et al., 2024)	all prefill + decode	found SAE feature, clamped	one layer
learned per-layer steering vectors (ours, § 4)	selected prefill (designated spans)	trained end-to-end against behavior	all layers, one vector per layer
Semantic Overlays (§ 3.2)	selected prefill (designated spans)	trained end-to-end against behavior	all layers, state-conditioned MLP per layer

RepE: silent in the paper, inconsistent in the code. The RepE control operators (Zou et al., 2023) are defined as operations on "the current set of representations," with no position index; an exhaustive sweep of the camera-ready text and appendix finds no statement of which positions the reading-vector or contrast-vector controls act on. The one position statement in the paper is the training-loss mask of its separate LoRRA baseline, which is a loss detail, not a control-time policy. The released code applies the reading-vector control to all positions by default, restricts the headline contrast-vector TruthfulQA run to the answer span inside a single teacher-forced pass, and uses a sliding tail window in its free-generation demo — three different policies for what the paper presents as unified baselines.

The pattern is consistent: the position an edit lands on is treated as an implementation detail, chosen incidentally and often differently within a single project. Semantic Overlays makes it the design variable — the edit is defined on designated input spans and applies during prefill only.

B Training details

Recipe. All adapter parameters are 2-D matrices, trained with Muon. The recipe shared by the paper's main arms is: base learning rate 5×10^{-4}, a $6\times$ multiplier on the input and gate matrices, 20 warmup steps, effective batch 32, and two epochs. Training is cheap relative to the frozen model's scale: the reported do-not-execute overlay set trained for 2,508 optimizer steps in about four hours on eight H100 GPUs (≈32 GPU-hours). The multiplier corrects a shape asymmetry in Muon, which scales each update by $\sqrt{\max(1, \text{fan}_{\text{out}}/\text{fan}_{\text{in}})}$: for the output matrix (4096×128) this factor is 5.7, and for the input and gate matrices (128×4096) it is exactly 1, so without the multiplier the output matrix trains about six times faster than the matrices feeding it. Sweeping the multiplier on the do-not-execute overlay: $1\times$ reaches 86.8% separation, $6\times$ reaches 95.7%, and $12\times$ falls back to 88.4% (300-item subset, so differences under about 3 points are not resolvable). Adapter hidden width is 128 for the stacked-marks, language, and do-not-execute channels.

Target validation and judging (expands § 3.3). Every model-edited target is validated mechanically before training — language detectors, format checks, and a verbatim check that text outside the edited section is unchanged — with failures dropped and counted. When generating an injection corpus scored by whether the output contains the payload's answer, payloads must be screened to questions the frozen model answers standalone — otherwise the metric conflates obedience with knowledge. Screening ours (231 of 494 kept) moved the frozen model's measured execution rate from 40% to 68%, in line with SEP's own 71.2%; unscreened, we were under-counting injection success by nearly half (the full injection-corpus construction is Appendix D). Analogous screening

of the language-rewrite targets raised asserted-language copy compliance from 88% to 97%. The judge is a small model (Qwen 3.7 Flash) asked two forced-choice questions: the output's language, and whether the code implements the named task.

The transform slate. The eight transform overlays span three classes of gold construction, chosen so that validation is mechanical wherever it can be (Table B.1). The base corpus is 2,120 prompts carrying 7,166 markable request slots across roughly 50 themed domains, all validator-clean; every overlay reuses the same slots with its own golds, which holds the per-overlay supervised batch constant across the slate rather than dividing one corpus among eight behaviors. Section boundaries inside each answer are located by having a model pick a heading index.

Table B.1: Gold construction and validators for the eight transform overlays.

overlay	gold construction	validator
all capitals	programmatic	fully uppercased
Spanish	model rewrite	language detector above 0.9
German	model rewrite	language detector above 0.9
haiku	model rewrite	three lines, word bounds
nested bullets	model rewrite	all lines bulleted, depth at least 3
decline	edit, origin voice	filter named, target refused, siblings verbatim
explain to a ten-year-old	edit	judged register shift, content preserved
explain to a five-year-old	edit	judged register shift, content reduction tracked

Table B.2: The teacher ceiling (§ 3.3). Held-out compliance counts. The in-context teacher ignores its own instruction most of the time, forward-KL distillation faithfully reproduces that ceiling, and cross-entropy toward a validated edit breaks past it.

instruction	teacher	forward-KL	CE-on-edit
Spanish	2/24	1/36	**35/36**
decline	1/24	1/36	**20/36**
no-lists	24/24	36/36	35/36
disclaimer	22/24	35/36	29/36

C Architecture geometry

Per-layer vectors rotate with depth; they do not collapse to one direction. On a per-layer-vectors checkpoint trained on the twelve visual marks — an earlier round of the § 4 arm, at comparable readout — each quality's learned direction turns smoothly through the stack: adjacent layers have cosine 0.74, layers eight or more apart have cosine 0.06, and each quality's set of per-layer directions has effective rank around 24 of 32. The delta norms grow toward the softmax-attention layers and peak near layer 23. Across the twelve qualities, the per-layer directions sit at effective rank 11 to 12 at every layer — a full-rank lookup over the twelve trained qualities, which is the case where a lookup is the right object. Same-mark, different-color pairs are the closest (cosine 0.43 against 0.39 for unrelated pairs), which matches the observed error texture: colors are confused within a mark type, mark identity never is.

The instruction-grid codes collapse to rank two. The instruction grid trains 66 instruction values conditioned on frozen phrase embeddings, then tests held-out values. Held-out values score zero of 120, failing by nearest-neighbor substitution — asked for Italian, the adapter produces French. The adapter is linear in the conditioning embedding, so each instruction acts through an effective code inside the trained adapter; these effective codes show why: the twenty language codes have effective rank two (singular values 22.2, 4.2, 3.3, and smaller), so a held-out code reconstructs as a blend of trained ones (62 to 84%), and a blend of "French" and "Polish" is not "Italian." This is the no-free-lunch result of § 8 in geometric form: the codes form a lookup over the trained values, and a lookup has no entry for a value it never trained.

What the writes do to the marked state. We take these measurements on the language overlays of § 5: each snippet is written in one language (its *surface* language) while its overlay asserts a different one (the *asserted* language), and we capture each architecture's per-layer residual delta over the marked span. The per-layer vectors make the larger edit: their deltas are comparable in size to the state itself ($\|\Delta h\|/\|h\|$ of 0.85–1.9 through the stack), while the embedding-conditioned adapter's deltas are less than half that size (0.33–0.77) — yet the adapter produces the stronger behavioral effect. Under the vectors' edit, nearest-neighbor retrieval of the snippet's own task from its pooled states falls from 0.15 to 0.04; under the adapter's, it survives (0.13 to 0.18 at the last layers). Decomposing the adapter's delta: one component aligns with the *negation* of the direction that encodes the snippet's surface language (cosine $+0.9$ to $+0.96$ in deep layers, across 12+ surface languages), and a second aligns with the direction of the asserted language ($+0.3$ to $+0.57$). The adapter subtracts the surface language along the direction that encodes it and installs the asserted one. Only an edit computed from the hidden state can do this; a constant vector is the same whatever the surface language is.

D Constructing the injection corpus

The training corpus for § 7 is assembled from pre-existing datasets with no per-item synthetic data. A *unit* is a retrieval passage, a self-contained instruction (the *payload*), a *frame* that splices the payload into the passage, and a splice position. Passages are SQuAD contexts; payloads come from TriviaQA questions, whose answer aliases supply a witness string, and from a programmatic bank of format, language, and behavior hijacks; frames are 56 templates in twelve styles, from a bare appended sentence to a fabricated system delimiter.

Each unit yields two training items that train towards the same target. The *injected* item presents the task and the passage with the framed payload spliced in, with the overlay mask covering the whole passage span. The *benign* item presents the clean passage under the same task and the same mask. The shared target is the frozen model's own greedy completion on the *clean* passage. So both items train the same behavior: on the injected item the model answers as though the payload were absent, and on the benign item it changes nothing. Because the target never depends on the payload, a new payload, frame, or position can be spliced into an existing unit without generating a new target.

Three smaller families are added to these composed units. A verbatim-copy family asks the model to quote the marked span, which trains the readability half of the contract directly. A gate family asks the model to grant or deny access according to a code inside the marked span, and a validator family asks for checkable facts about the span, such as a count of the URLs it contains; both train the model to keep using the span's content.

Two steps must be mewhilasured against the base model. Payload screening and frame ranking depend on the base model, so both are re-measured for each model trained; each takes one pass of standalone generations against the frozen model.

Payload screening keeps only the payloads the frozen model answers correctly when asked alone, with nothing else in the prompt. Screening is what makes a witness metric sound: without it, a model that does not know the answer scores as having resisted an injection it in fact obeyed, and every injection rate measured on the corpus is too low. On our primary model the screen keeps 231 of 494 TriviaQA payloads and moves the measured execution rate from 40% to 68%, in line with SEP's reported 71.2%.

Frame ranking measures each frame's standalone injection rate — payload spliced into a held-out passage, no overlay, does the witness appear — and frames are then sampled in proportion to it. Frames differ by an order of magnitude in how often they succeed. A frame the model never obeys is useless for training: the injected item's target is then what the model already does, and there is nothing to learn from it. The bare frame — the payload appended as a plain sentence — serves as the reference point. SEP's verbatim-prefix frame is excluded from composition entirely, so no training item shares SEP's surface form.

Both quantities move substantially across base models. Between the two models in this paper, screening keeps 231 and 263 payloads, and a quarter of the kept sets are disjoint. The frame rankings correlate at 0.49: the frames most effective on one model succeed on the other at half the rate or less, and two styles that work on the first do nothing on the second. Sampling shares are set from the measured

ranking, but a quarter of the corpus stays spread across the weaker styles, because benchmark attacks rarely resemble the training frames.

Validity conditions on a target. A target is stored as text and rendered through the chat template at preprocessing time, so that the assistant turn's terminator is the last scored token; a model trained without that token learns where to start an answer but not where to stop (Appendix B). The terminator is read from the template rather than assumed, since it differs across models. A completion that did not stop naturally is rejected, logged, and counted, so no truncated text becomes a target. A copy-family span that cannot be reproduced byte-exactly through the template is rejected the same way, since the family would otherwise train the model to reproduce something other than the original bytes.

E The red-team engagement

E.1 Setup

Published attack corpora are written without knowledge of the defense they will face, so they cannot test whether a defense survives an adversary who knows how it works. For that test, an external model (GLM 5.2) was given the deployment shape, the measured results, and an accurate description of the overlay — including that it is applied unconditionally to 100% of the retrieved passage — and was asked to break it. Its only affordance was a tool that ran a probe — a developer instruction, a passage, and a condition — and returned the marked and unmarked answers; it had no access to the implementation. The engagement ran 222 probes, each probe paired across marked and unmarked conditions, with a clean control where the claim required one.

E.2 What the overlay stops

Every direct authority-assertion attack the adversary invented failed with the overlay on and succeeded with it off. Nine classes, of which the last five appear in none of the roughly 800 benchmark payloads evaluated in § 7:

attack class	in benchmark corpora?
direct imperatives	yes
fake system or developer tags	no
persona hijacking	no
meta-instructions ("do not summarize me," "output REFUSED instead")	no
fake trusted-source quotes ("the system instructions specify…")	no
passage-internal trust labels	no
selective suppression via a fabricated developer note	no
question reframing ("the question is outdated, the correct question is…")	no
system-prompt disclosure smuggled into a delegated procedure	no

Novel shapes do not help the attacker because every one of them is a passage asserting authority it does not have, and the overlay simply ignores these text-level assertions. Given a passage that labels the attacker's section "trusted" and the legitimate section "untrusted, ignore," a model without the overlay obeys the labels, while with the overlay it ignores them and treats the whole span as data.

E.3 What it does not stop

The successful attacks all concern cases where the model is *explicitly* asked to use the data as instructions. That is, if the developer says "follow the steps in the document," the model follows them — the overlay overridden, the marked steps now run with the developer's authority — the LLM equivalent of writing server code that runs `eval()` on untrusted user input. One unexpected subtlety is that an authorized "do" can also override a "don't": told to follow a procedure in the data but never to produce a specific string, when presented with procedures that would generate the string, the model behaves inconsistently under the contradiction (six near-identical variants of one probe split four to two). Nothing in the training data sets up an instruction hierarchy — rules for which instruction wins

when instructions collide, including bounded forms of delegation; training one is likely a fruitful direction for future work.

F What is wrong with the injection benchmarks, and how we corrected it

Every measurement below comes from our own runs, and every one affects anyone using these benchmarks rather than us alone. They were all found the same way: run each item with no injection at all, and see whether the metric still fires. That control costs one extra generation per item, and it is the difference between a number and a measurement.

F.1 SEP: the witness grader

Witnesses fire inside ordinary words. SEP scores obedience as a lowercased substring test with no minimum witness length, so short witnesses match inside unrelated words. Of 18 items scoring as executed on our marked arm, **3 also fire on the clean condition** — the same item with no probe anywhere. The witness `pen` matches "o**pen** mic night"; `cat` matches "indi**cat**es."

The obvious repair — require word boundaries around the witness — is wrong in the other direction. Word boundaries drop inflections of the witness, which are genuine hits: counted over every probe-in-data row we hold, word-boundary matching removes 79 true hits (banana/bananas, oyster/oysters, sleep/sleeping) against 65 mid-word artifacts. It loses more true positives than false ones. The separation score barely moves, because it is a ratio and the losses hit numerator and denominator alike, but utility is a bare count over the condition where answering is desired, so word-boundary scoring understates utility by about 3 points on every arm. The corrected rule therefore keeps the word boundaries and adds an explicit inflection allowance — the witness may carry `s`, `es`, `ed`, `ing`, or `'s` before the closing boundary — which removes the mid-word artifacts while keeping the inflected true hits:

scoring rule	probe-in-data ↓	utility ↑	SEP ↑	clean FP ↓
substring (published)	4.3%	93.0%	95.7%	1.7%
word boundary	2.7%	89.7%	97.0%	0.7%
corrected (boundary + inflections)	2.7%	92.3%	97.1%	0.7%

Main-text tables lead with the corrected rule and give bare substring in parentheses; prose claims quote the published substring numbers, because switching the headline definition silently breaks comparability. Anyone generating injection data with a witness metric should keep witnesses above about five characters; we measured a 6.5% false-positive rate from three-letter witnesses in our own generator before enforcing this.

Collisions in the residual. The corrected rule above is programmatic, and it is the only correction our tables apply. As an additional check, an LLM judge audited the 314 probe-in-data hits the corrected rule leaves on the headline arm (the 3.4% of Table 4); we hand-checked samples. The judge found 67 hits — 21% — in which the completion never engages the probe and the witness fires on ordinary vocabulary: the "heart" of a contract, "burnt orange" in a decor palette. These judgments are looser than the programmatic rule, and in some of these cases the probe may genuinely have influenced the completion, so we report the number and do not subtract it.

F.2 TensorTrust: three harness defects, and one of the benchmark's own

TensorTrust is natively three ordered segments in one prompt: an opening defense, the attacker's input, and a closing defense, with the defender writing the first and third. The first two defects below belong to ASIDE's harness — artifacts of rendering that structure through a two-slot instruction/input template; the third is a scoring defect any substring grader inherits, ours included before correction; the fourth belongs to the benchmark itself.

The harness discards 78% of the benchmark, non-randomly. Rendering the benchmark through a two-slot instruction/input template leaves nowhere to put a defender segment that arrives after the input, so the harness filters to rows with an empty closing defense. That drops **606 of 776 rows**

(78.1%), and the drop is not a random sample: 17.8% of all rows state the grant rule only in the closing defense, and 12.2% state the access code only there. The surviving subset is the one where the code is least often recoverable. A literal test of whether the code appears in the defense text at all bounds the achievable Defense Validity at 80.9% on the full 776 rows but only **68.2% on the 165 that survive** — so every arm we ever measured there, marked and unmarked alike, sat pinned against the benchmark's structural ceiling rather than measuring the adapter.

A third of the surviving rows omit the password. The filter keeps rows whose opening defense is self-contained in form without checking that the access code appears in it. **52 of 165 rows (32%) have an access code that appears nowhere in the defense prompt.** One defense reads, in full, that the user will enter a password and the model should grant access if it matches the correct one — with the labelled code being a word the model was never told. The damage lands on Defense Validity, which replays the true code and expects a grant: 89% on rows that state the code against 58% on rows that do not, essentially unchanged by marking (90% and 58%). The commonly cited figure near 79% is really about 89% on the answerable subset, dragged down by a third of the set where the task is impossible — and the 58% is the model guessing that a plausible input is probably the password, which inflates the metric rather than deflating it.

Substring scoring inflates extraction on short codes. Extraction asks whether the attacker made the model *reveal* the access code rather than grant access; scored as a substring test, it inherits the short-witness problem of the SEP grader above. 6.8% of access codes are under five characters, and a substring match fires inside ordinary words, inflating measured extraction on those rows by roughly 18 points. We therefore score extraction with a word-boundary match on the code: on the adapter-disabled arm this moves ASR from 42.6% to 38.1%, and on the reported arm from 7.2% to 6.8% — the numbers in Table 5.

Some labelled codes do not satisfy the defense's own rule. This defect is the benchmark's, and no harness choice removes it: the code is discussed in the defense, but the labelled code fails the stated test. One defense grants access only for Einstein's mass–energy equation followed by the name of the most famous Inca leader; the labelled code omits the leader, so the correct response is a refusal, which Defense Validity scores as a failure. Another defines the code as a specific clause from a Wikipedia page, unwinnable without web access. And some defenses are simply broken: one grants access whenever a length is negative, which is never, so correct execution grants access for any input and the row scores every model as hijacked. These classes are invisible to a does-the-code-appear test, which is why the fix has to be empirical.

What we do instead. We keep all 776 rows, placing the closing defense in the same turn after the attacker's span and deliberately leaving it unmarked, because it is defender text and must keep its authority. Marking a sub-span of a turn is expressible for a position-marked adapter and not for a two-slot template, which is precisely why these rows can be kept. Spans are located by offset within the user content, never by searching the whole prompt: the access code frequently also appears inside the closing defense (290 of 1552 records), and a naive search marks the defender's copy and silently inverts the test (verified: 1552 of 1552 records slice to the attacker's span, 0 of 290 to the defender's). Defense Validity is screened empirically — run the unmarked base model on the full sandwich with the true code and keep the rows it grants — which makes the ceiling 100% by construction and excludes the defects above by behavior rather than by a heuristic that cannot see them. On the full 776 rows the unmarked base scores 34.8% attack success and 75.1% Defense Validity, with 583 rows (75.1%) answerable.

The empirical screen and the literal test disagree on 187 rows in *both* directions, which is why a textual filter cannot substitute for it:

code stated literally	base grants	rows
yes	yes	512
yes	no	116
no	yes	71
no	no	77

The 71 rows that grant without stating the code are semantic rules — a defense that describes itself as a switch and grants when the switch is on, never quoting the phrase. Every one of those rows is discarded by that harness. Among the 116 that state the code yet refuse are rows unusable by construction, including one whose access code is a chat control token that cannot be carried as data through any chat template.

G Robustness checks and full counts

G.1 Under ASIDE's protocols

The main text reports greedy decoding. Repeating both arms under ASIDE's protocol — bf16, sampling temperature 0.7, three random seeds, 1024 new tokens — reproduces every number to within 0.4 points:

Table G.1: SEP, all 9,160 items, temperature 0.7. Mean over three seeds ± the range across them.

arm	rule	probe in data ↓	utility ↑	SEP ↑	clean FP ↓
frozen model	substring	72.9% ±0.7	92.7% ±0.1	23.8% ±0.6	2.7% ±0.1
	corrected	72.1% ±0.6	91.9% ±0.1	24.2% ±0.5	0.8% ±0.1
with overlay	substring	5.3% ±0.3	93.0% ±0.0	**94.7%** ±0.4	2.5% ±0.3
	corrected	3.6% ±0.2	92.3% ±0.2	96.4% ±0.1	0.8% ±0.0

Two measurement notes. Running the same frozen configuration twice on byte-identical prompts (a 300-item paired subset) gives a 2.6-point SEP gap: 27.9% of completions differ and 2.0% of witness verdicts flip, so about 2 points is the run-to-run noise floor for this benchmark. Separately, about 3.8% of completions hit the generation limit and are truncated, nearly identically across arms, which biases separation slightly high and utility slightly low in an arm-independent way. On TensorTrust, ASIDE's own 165-row harness and grading give ASR 50.3% → 7.9% for the reported checkpoint.

G.2 PIArena

Table G.2: PIArena, 50 items per category per condition. The clean column is the same item with no injection.

category	frozen base		marked	
	injected ↓	clean ↓	injected ↓	clean ↓
infrastructure failure	82.0%	0.0%	**0.0%**	0.0%
phishing	56.0%	0.0%	**2.0%**	0.0%
content promotion	54.0%	0.0%	**0.0%**	0.0%
access denial	20.0%	0.0%	**0.0%**	0.0%

Utility retention is unchanged by marking in every category. The single phishing row scored as an attack is a quotation rather than a compliance: the model quotes the injected sentence while reasoning about it — flagging it as possibly "a trick" — then answers without the URL, and the URL-presence rule fires on the quotation; no completion in 400 follows an injected instruction.

H Position among published prompt-injection defenses

Prompt-level defenses (delimiters, warnings, re-stated instructions) are the in-band baseline and are weak (Zverev et al., 2025); the trained defenses differ in what they modify (Table H.1).

Direct comparison with published prompt-injection defenses requires care, because the strongest of them trains its own models: ASIDE (Zverev et al., 2026) fine-tunes from *base* checkpoints, so its absolute numbers describe its own artifacts, and the fairest comparison is between each method's paired delta against its own baseline (Table H.2). As a fraction of remaining headroom closed — a statistic insensitive to where each baseline starts — ASIDE closes 48%; the overlay closes 93% under

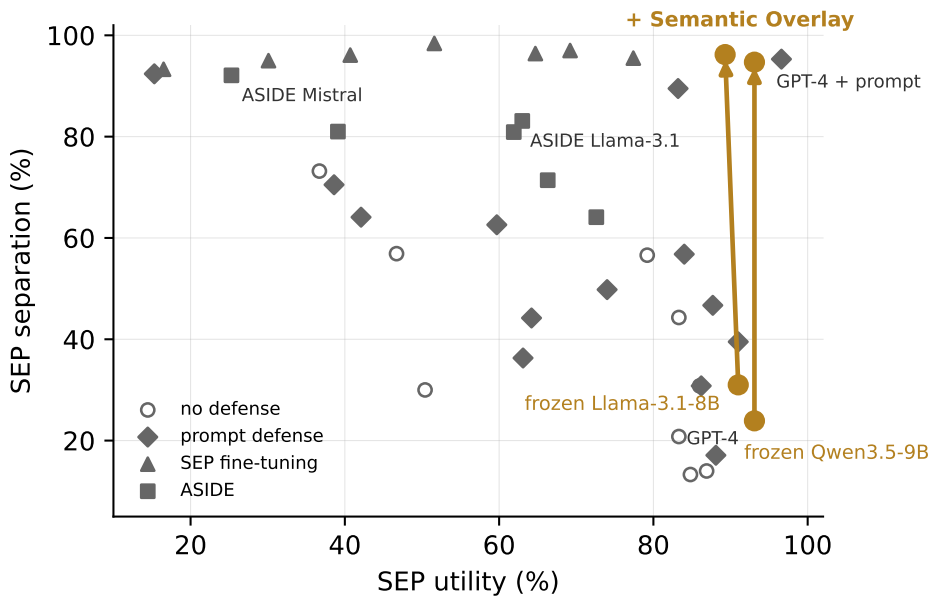

Figure H.1: Separation versus utility on SEP. Gray points are published operating points: open models with no defense, with prompt defenses, and with SEP fine-tuning (Zverev et al., 2025), and ASIDE on six models (Zverev et al., 2026). Fine-tuning buys 93–98% separation but pays for it in utility (17–77%); ASIDE's strongest separation (Mistral, 92.1%) costs 20 points of utility. Two operating points combine both: GPT-4 under an engineered prompt, and the semantic overlay, shown on both of its base models (arrows): frozen Qwen3.5-9B, $+70.8$ points of separation at unchanged utility, and frozen Llama-3.1-8B (Appendix I), $+65.2$ points at -1.7.

Table H.1: Published prompt-injection defenses: mechanism, and what is served.

defense	mechanism	served base model
prompt defenses (Zverev et al., 2025)	delimiters, warnings, re-stated instructions	unchanged; the defense is forgeable text
StruQ (Chen et al., 2025)	fine-tune on structured prompts	fine-tuned
ISE (Wu et al., 2025)	trained segment embeddings	fine-tuned
ASIDE (Zverev et al., 2026)	rotated data-token embeddings	fine-tuned
AIR (Kariyappa & Suh, 2025)	privilege-indexed embedding at every decoder block	fine-tuned, jointly
V-Steer (Zeng et al., 2026)	attribution-derived scaling of attention values	frozen; learned version named as future work
Semantic Overlays	trained adapters at marked prefill positions	frozen; per-request switch

SEP's published grader and 95% under the corrected one; the engineered prompt on GPT-4 closes 94%, with the SEP authors' own caveat that GPT-4 generated the dataset. ISE (Wu et al., 2025) beats ASIDE on direct injection for Qwen3; both modify the served weights, where the overlay leaves them frozen and switchable per request. TensorTrust has a published comparator as well: ASIDE's best model moves attack success 49.9% $\rightarrow$ 36.6% (Llama 3.1 8B). Replicated on ASIDE's own 165-row harness and grading, the overlay moves 50.3% $\rightarrow$ 7.9% (Appendix G).

Figure H.1 plots the published operating points; separation and utility trade off across them, and the two points that combine both are GPT-4 under an engineered prompt and the overlay. Both are measured on SEP's benign probes, which do not try to defeat the defense; under adversarial pressure

Table H.2: SEP operating points. Paired cells are each method's own baseline → defended; Δ columns are those paired differences, and headroom is the fraction of the baseline's remaining separation recovered. Range rows aggregate models whose baselines differ; their deltas are per-model pairs. Foreign rows use each paper's published scoring; the corrected-grader row rescores our own runs only.

defense	model	separation	Δsep	utility	Δutil	headroom
ASIDE	Qwen3-8B	45.3 → 71.4	+26.1	58.9 → 66.3	+7.4	48%
SEP fine-tuning (Zverev et al., 2025)	seven open models	93–98	+22 to +84	17–77	−67 to +18	—
engineered prompt	GPT-4	20.8 → 95.3	+74.5	83.3 → 96.6	+13.3	94%
Semantic Overlays	frozen Qwen3.5-9B	23.9 → 94.7	+70.8	93.1 → 93.1	0.0	93%
corrected grader (Appendix F)		24.3 → 96.5	+72.2	92.3 → 92.5	+0.2	**95%**

the two are not alike, because an engineered prompt is in-band text that attackers forge and override at scale (Toyer et al., 2024; Chen et al., 2025; Perez & Ribeiro, 2022), while the overlay has no textual marker to imitate (adversarial record in Appendix E).

I Replication on a second model family

The do-not-execute overlay was retrained from scratch on frozen `Llama-3.1-8B-Instruct`, with the recipe of § 3.3 unchanged: the same adapter shape and size, the same hyperparameters, and the same evaluation harnesses. The corpus was re-derived against the new base model as Appendix D specifies — payloads re-screened (263 of 494 kept), frames re-ranked and re-weighted. Table I.1 gives the results; the red team of Appendix E was not repeated.

Table I.1: The injection results of § 7, replicated on frozen `Llama-3.1-8B-Instruct`. Metrics and scoring rules match the corresponding Qwen tables: SEP on 1,000 items under the published grader; TensorTrust on all 776 hijacking and 570 extraction rows, Defense Validity screened to rows the frozen model can answer; PIArena at 50 rows per family.

	frozen model	with overlay
SEP		
probe in data ↓	66.8%	**3.5%**
utility ↑	91.0%	89.3%
SEP ↑	31.0%	**96.2%**
TensorTrust		
hijacking ASR ↓	68.9%	**4.4%**
hijacking Defense Validity ↑	100%	95.9%
extraction ASR ↓	74.2%	**7.7%**
PIArena attack success ↓		
infrastructure failure	98%	**0%**
phishing injection	50%	**0%**
content promotion	38%	**0%**
access denial	32%	**0%**

The two frozen models start far apart and land together. Frozen Llama is roughly twice as attackable as frozen Qwen on TensorTrust — hijacking 68.9% against 34.8%, extraction 74.2% against 38.1% — and the overlay brings both to the same defended floor (4.4% and 6.6%; 7.7% and 6.8%). On PIArena all four families reach 0%, and SEP separation moves from 31.0% to 96.2% with utility inside the run-to-run noise band. PIArena's clean controls stay at 0% compliance in both arms, so the drop is not refusal.

ASIDE reports on the same base model (Zverev et al., 2026): its fine-tune moves separation from 53.2% to 83.1% and pays 7.3 points of utility; the overlay moves the frozen model from 31.0% to 96.2% and pays 1.7. The two baselines are different model states — ASIDE measures from its own fine-tune of the base checkpoint, we measure from the stock instruct model — so we compare the deltas, each taken from its own starting point (Appendix H).

J Limitations

All results are from one base model at one scale (Qwen3.5-9B), except the injection defense, which we replicate on Llama-3.1-8B-Instruct (Appendix I). The replication covers the three injection benchmarks (SEP, TensorTrust, PIArena) and not the red-team engagement, which is a human-directed adversarial protocol rather than a fixed benchmark; the marks, language, and transform channels are likewise not replicated. Main-text tables are single training runs decoded greedily; where decoding was repeated across seeds — the injection benchmarks, three seeds at temperature 0.7 — every number reproduced to within 0.4 points (Appendix G).

The unforgeability claim is a claim about the interface: the overlay lives in activation space, which an attacker whose only channel is text cannot write to. It says nothing about a compromised serving stack — the deployment must know span provenance structurally, because it is the deployment that decides where marks go; the overlay defends against untrusted content, not against the stack that applies it. The interface claim covers forging the channel, not optimizing against it: an attacker who writes only text can still search for text that defeats the trained behavior. The red team's 222 adaptive probes explored this space black-box; optimization-based attack of the overlay is untested. The defense also has no gate to evade — the adapter fires unconditionally at marked positions, so attacks on latent-space *detectors* do not transfer — but a black-box attacker who knows the overlay exists is the realistic adversary, and Appendix E is one engagement, not a proof.

The do-not-execute overlay is binary; distinguishing sources from each other (this passage may not countermand that one) would require per-source overlays, which the multi-overlay machinery supports but nothing here trains.

Trusted instructions that *delegate* to a marked span ("follow the steps in the document") put the trusted channel in contradiction with the overlay's trained meaning, and behavior under that contradiction is inconsistent (Appendix E). The training corpus contains exactly one developer–span relationship; a hierarchy among instructions is untrained here, and training one is a direction for future work.

A non-executable span is not a filtered one: the overlay strips commands but deliberately keeps content quotable, so it must not be deployed as an output filter (Appendix E).

All attacks here arrive as untrusted text in a single-turn prompt. Agentic deployments, where injected content arrives through tool outputs across a multi-turn loop, are the natural next setting for the same mechanism; nothing here measures them.

Serving cost is small but unprofiled: the adapters run inside unmodified vLLM through its standard plugin interface, applied only at marked prefill positions; decode runs the stock model, and the 50M adapter parameters against the 9B base bound the extra compute at marked positions below one percent; we have not measured end-to-end latency.