When Models Know but Can't Linearly Say: Relational Encoding Limits Linear Vulnerability Classification in Code LLMs

Rui Melo[1,2*] **André Catarino**[1,2*] **Cláudia Mamede**[1,2]
Corina Pasareanu[1] **Rui Abreu**[2]

[1]Carnegie Mellon University, Pittsburgh, PA, USA
[2]Faculty of Engineering, University of Porto, Portugal

Abstract

Large language models can detect vulnerable code, yet the form of vulnerability encoding in their representations remains unclear. We analyze 8 code models spanning 6.7B–15B parameters on C programs using three datasets (DELTASECOMMITS, SVEN, and PRECISEBUGS), comparing two diagnostic tasks on the same representations: identifying *what patterns* the code instantiates, and determining *whether it is vulnerable*. Models encode the first with high fidelity. Under the studied paired C-code datasets and linear-probing setup, vulnerability information is more accessible as a *paired relational signal*—whether code A is riskier than code B—than as a globally separable categorical label. This structural difference has implications for classification: the model reliably ranks which code is riskier but standard classifiers on isolated samples achieve limited accuracy because vulnerable and secure samples form overlapping distributions in representation space.

1 Introduction

Manual code review is labor-intensive and does not scale. Continuous Integration and Deployment (CI/CD) pipelines require automated vulnerability detection to reduce time-to-patch and improve coverage of codebases. Neural-based classifiers have been a long-standing approach to this problem (Zhou et al., 2019) that falls short of production deployment (Ullah et al., 2024; Khare et al., 2023; Fu et al., 2023; Ding et al., 2024).

Recent work demonstrates that large language models (LLMs) can autonomously discover zero-day vulnerabilities in real-world systems (Anthropic et al., 2026; Anthropic, 2026). This raises a crucial question: models can discover vulnerabilities, yet classification approaches are expected to fail. *How do these models represent code safety, and why does their representation resist standard exploitation?*

The core question is not just *whether* models encode safety information, but *how*. Classification-based approaches assume vulnerability is represented as an absolute categorical property.

We investigate through two diagnostic tasks on identical code representations: one measures *what vulnerability patterns the code instantiates*; the other measures *whether the code is vulnerable*. Both probe the same model but measure different aspects of learned structure. The model encodes vulnerability patterns with high fidelity, yet absolute safety classification rather weakly—a critical gap that motivates deeper investigation.

We find that, under the studied paired C-code datasets and linear-probing setup, vulnerability information is more accessible as a **paired relational signal**—version B is more vulnerable than version A—than as a globally separable categorical label of the kind standard classification assumes. This relational direction achieves 87–88% per-pair alignment consistently across layers, yet linear probes achieve only modest categorical accuracy because vulnerable and secure samples form *overlapping distributions* in the representation space. In other words, the model reliably determines which code is riskier within pairs, but linear classification achieves limited performance.

This finding reconciles a puzzling observation: models discover zero-day vulnerabilities, yet classification approaches are expected to fail. Models have learned to represent vulnerabilities but organize them in a form that standard linear classification cannot readily exploit. We characterize this representational mismatch across multiple model architectures and scales and discuss implications for detection systems that leverage this structure rather than fighting it.

*Equal contribution. rmelo@cs.cmu.edu

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 26461–26497
October 24-29, 2026 ©2026 Association for Computational Linguistics

2 Related Work

Vulnerability detection. Early supervised approaches for vulnerability detection (e.g., Devign (Zhou et al., 2019), CodeBERT variants (Croft et al., 2023), transformer-based models (Thapa et al., 2022), SecureFalcon (Ferrag et al., 2025)) train encoder-based models on paired datasets, typically achieving 0.8–0.95 AUROC on in-distribution test sets. Recent comprehensive evaluations (Ullah et al., 2024; Khare et al., 2023) demonstrate that general-purpose code LLMs struggle with absolute vulnerability classification, with only moderate AUROC on categorical tasks, yet show promise in ranking-based and reasoning-focused evaluation paradigms (Ding et al., 2024).

However, supervised approaches face fundamental brittleness: post-hoc analysis reveals models exploit dataset artifacts and spurious correlations, and performance degrades sharply on out-of-distribution data, as illustrated by Ding et al. (2025). This motivates a shift toward representation-based approaches that ask not *how to classify* but *what vulnerability information LLMs encode*—and critically, how that information is geometrically organized in activation space. This work provides mechanistic insights into this organization, explaining both why vulnerability signals are difficult to access and how representation-level structure determines what downstream tasks can extract.

Pairwise preference modeling and ranking. Recent work frames the alignment problem through pairwise preference modeling: Direct Preference Optimization (Rafailov et al., 2023) trains LLMs directly from pairwise comparisons of completions without explicit reward modeling. This preference-learning paradigm, rooted by Bradley–Terry (Bradley and Terry, 1952) and Thurstone (Mosteller, 1951), has become foundational to modern fine-tuning. Recent code generation benchmarks like BigCodeBench (Zhuo et al., 2025), SWE-bench (Jimenez et al., 2024), and LiveCodeBench (Jain et al., 2024) make pairwise and ranking-based evaluation more tractable.

Our observation that vulnerability is encoded through relational structure aligns naturally with this pairwise paradigm: we demonstrate that LLMs encode vulnerability as a comparative signal suitable for ranking, rather than as a categorical classification feature readable from isolated samples. In contrast to absolute-threshold binary classifiers that struggle with overlapping distributions, ranking-based approaches (e.g., scoring code by its position along the vulnerability direction, or using pairwise preference objectives during fine-tuning) are potentially better aligned with how the model actually encodes vulnerability. The identified vulnerability direction could serve both as a diagnostic for understanding what representations the model has learned, and as a basis for preference-learning approaches that directly optimize for ranking. This direction could be leveraged as an implicit reward signal for steering code generation toward safer outputs (Gao et al., 2024), complementing explicit reward modeling approaches (Yuan et al., 2024).

Linear directions for safety and semantic properties. The analysis of intermediate representations has been largely driven by linear probing. Alain and Bengio (2017) introduced the use of linear classifiers on frozen representations as a diagnostic step for assessing whether certain information is accessible in the learned features. Ian Tenney and Pavlick (2019) demonstrated that pretrained transformers have layer-wise depth-related concepts that become progressively more specialized. Recent work has identified that high-level properties like factual knowledge (Meng et al., 2022) and task preferences (Zou et al., 2023) can be extracted and manipulated as linear directions in activation space. Our work extends this methodology to vulnerability: we identify the vulnerability direction as a stable linear direction across layers and characterize its geometric properties and cross-family transferability.

LLMs and code security: representation-level findings. Recent work demonstrates that code LLMs encode security-relevant information that can be extracted and used for steering (Yousefpour et al., 2025). He and Vechev (2023) show that property-specific control vectors steer generation toward secure code. Yu et al. (2025) propose dynamic mixtures of linear corrections applied during decoding. Huang et al. (2026) deploy learnable security steering matrices at the embedding layer, using contrastive decoding to balance functional and security objectives.

Probing, representation engineering, and interpretability. Probing classifiers assess what properties LLMs encode by training simple predictors on frozen representations (Tenney et al., 2019; Hewitt and Manning, 2019). Hewitt and Liang (2019a) establish that control tasks are essential to

distinguish genuine encoding from spurious correlations; our within-language and within-CWE experiments satisfy this criterion by construction. In the code domain, recent work on vulnerability-specific probing (e.g., Ribeiro et al. (2026) on code correctness as linear directions) demonstrates that property-specific signals can guide candidate ranking. However, the interplay between representation structure (is the signal concentrated or diffuse?) and downstream task design (should we aggregate across positions or read from a single token?) remains underexplored. Representation engineering approaches (Zou et al., 2023) identify high-level concepts as directions in activation space for diagnosis and steering. Causal-tracing methods (Meng et al., 2022) establish which intermediate activations causally contribute to outputs. This work builds on these insights by characterizing the vulnerability signal as a stable direction in activation space and demonstrating that its geometric properties determine accessibility for downstream tasks.

Complementary system-level and pattern-based detection approaches. Our representation-level findings complement, rather than compete with, other vulnerability detection strategies. Graph-based methods such as GRACE exploit program structure and data-flow information to improve detection (Lu et al., 2024), while similarity-based approaches such as VulSim leverage shared vulnerability patterns across code embeddings (Shimmi et al., 2024). Dataset-driven studies also show that curated vulnerability corpora can support secure code generation and vulnerability mitigation (Wang et al., 2023). More broadly, these lines of work highlight that vulnerability detection benefits from structural, relational, and data-centric signals, alongside benchmark-driven evaluation (Peng et al., 2025). In this context, our characterization of how LLMs encode vulnerability suggests that downstream systems may be especially well suited to ranking-based triage, code-review prioritization, and pairwise comparisons rather than only absolute binary classification thresholds.

3 Methodology

Models We study 8 LLMs spanning three families: Qwen2.5-7B-Instruct (Hui et al., 2024), CodeLlama-7B-Instruct (Rozière et al., 2024), StarCoder2-7B (Lozhkov et al., 2024), DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024), Qwen2.5-Coder-7B-Instruct, Qwen2.5-14B-Instruct, CodeLlama-13B-Instruct, and StarCoder2-15B.

Dataset We focus our analysis on C code, using three datasets to test whether the relational encoding pattern is specific to one curation strategy or replicates across dataset scale and construction methodology: (1) DELTASECOMMITS (Melo et al., 2026), a curated dataset of real-world vulnerability-fixing commits from OSV and NVD filtered to C code ($n = 1{,}618$ paired samples, CWE-identified). This paired structure controls for author style and codebase idioms; pairs differ primarily in the patch applied to address the reported vulnerability. (2) SVEN (He and Vechev, 2023) (Semantic Vulnerability Extraction Network; $n = 423$ paired samples across 6 CWEs, synthetically curated by semantic transformation). (3) PRECISEBUGS (He et al., 2023) (real-world fixes from open-source repositories; $n = 4{,}101$ paired samples across 9 CWEs, extracted from actual commits).

Activation Pooling and Extraction We extract residual stream activations from the different LLMs. For binary probing, we extract at the last token position as independent samples. For distributional analyses, we use per-pair differences $\delta_i = v_i - s_i$ to cancel commit-level confounds. For mean-token pooling, we average across all tokens.

Probe Design: Why Linear and Shallow We use linear logistic regression probes, not out of simplicity but precision. A complex probe obscures the interpretation: high accuracy could reflect the classifier computing a nonlinear combination of weak scattered signals rather than the model encoding vulnerability directly. Hewitt and Liang (2019b) demonstrate that overparameterized probes memorize control task labels, conflating true encoding with overfitting. Linear probes, by contrast, succeed only when information is genuinely accessible and linearly organized in the representation. This follows Park et al. (2024), which shows that high-level concepts in LLMs are structured as linear directions in activation space. A linear probe tests whether the model has organized vulnerability information as a linear direction; if vulnerability signals are so organized, the probe will recover them; if not, the low AUROC reflects the signal structure, not the probe's capacity.

Statistical Evaluation and Metrics Standard fold stratified CV prevents label leakage across folds, but paired datasets can leak if both members of a pair are in different folds. We imple-

mented pair-stratified CV (both members in the same fold) to eliminate within-pair contamination. Binary probing resamples at sample level; paired-delta analyses resample at pair level.

We report point estimates for all primary metrics (per-pair alignment, AUROC, cross-family transfer). Cross-model consistency of per-pair alignment is quantified using Pearson correlations (mean $r = 0.921$ across the three model pairs). The pair-stratified cross-validation structure enables computation of fold-level confidence intervals; reported point estimates are fold-wise means, and fold-wise variance measures are available for uncertainty quantification. Cross-family transfer rates are reported as mean transfer across all family pairs, with per-pair transfer values provided in heatmaps to show variance across transfer directions.

Cross-validation, normalization, and dimensionality. We distinguish between preprocessing pipelines for different analyses to ensure reproducibility and address potential artifacts:

Direction computation (raw activations). The per-pair alignment metrics and cross-layer direction similarity are computed on raw activations *without* normalization or dimensionality reduction. The vulnerability direction is: $\hat{d}^{(l)} = \frac{\bar{\mathbf{v}}^{(l)} - \bar{\mathbf{s}}^{(l)}}{\|\bar{\mathbf{v}}^{(l)} - \bar{\mathbf{s}}^{(l)}\|_2}$, where $\bar{\mathbf{v}}^{(l)}$ and $\bar{\mathbf{s}}^{(l)}$ are layer-wise means computed on training data. Per-pair alignment is the fraction of test pairs where $(\mathbf{v}_i - \mathbf{s}_i) \cdot \hat{d} > 0$. No PCA or z-score normalization is applied, ensuring the reported 87–88% alignment and 0.993 cosine similarity reflect geometric properties of the raw representation space.

Linear probing (PCA-reduced, per-fold). Classification probes use dimensionality reduction per fold. Each fold's training data undergoes: (1) Training activations are z-score normalized (StandardScaler); (2) Z-scored training data are projected to 50 PCA components, fitted on training data only; and (3) Test activations are transformed using the training fold's scaler and PCA. Pair-stratified cross-validation prevents within-pair leakage. Such is independent of direction computation, analyzing the signal in a reduced space to control for noise while maintaining test separation. We perform sensitivity analyses (Appendix A): PCA-50 vs PCA-100 differ <0.01 AUROC.

Vulnerability direction computation and test leakage. The vulnerability direction is computed as the normalized mean difference per layer. We employ two complementary approaches: *Per-fold directions* (computed on training data within each fold) prevent test leakage and enable per-fold cross-validation of per-pair alignment. *Per-family directions for cross-family transfer* are computed on training data per fold per CWE family, ensuring that test pairs used to evaluate transfer are unseen during direction fitting. This per-fold-per-family approach assesses whether the learned direction captures universal vulnerability properties transferable across CWE families, while maintaining strict train/test separation. Per-pair alignment for all analyses is measured on held-out test data unseen during direction fitting. All analyses use pair-stratified splits (both vulnerable and secure members assigned to the same fold) to prevent within-pair label leakage.

4 Functional vs. Security Encoding

4.1 Strong Functional Encoding: Within-Language CWE-Type Separation

We first establish that the model encodes different vulnerability-pattern families with high fidelity by testing whether it can distinguish different CWE types within a single language (e.g., memory-safety violations vs. injection attacks). This is a functional encoding task: we probe only on vulnerable samples, so the classifier distinguishes code patterns rather than vulnerability status.

We run pairwise binary CWE-type probes by extracting residual stream activations from vulnerable samples across all token positions, applying PCA-50 projection, and training a logistic regression classifier in 5-fold cross-validation. Figure 1 shows AUROC matrices for DELTASECOMMITS.

Results: Strong CWE-type separation. CWE-type probing shows robust functional encoding: across the eight primary vulnerability categories shown in Figure 1, most pairs exceed AUROC 0.85. This pattern of strong functional CWE separation replicates consistently across all 8 models spanning 6.7B to 15B parameters: Qwen2.5-7B-Instruct and Qwen2.5-14B-Instruct, CodeLlama-7B-Instruct and CodeLlama-13B-Instruct, StarCoder2-7B and StarCoder2-15B, DeepSeek-Coder-6.7B-Instruct, and Qwen2.5-Coder-7B-Instruct (see Appendix E for per-model heatmaps). Mean pairwise AUROC across the 7B models averages 91.4%, while larger models (13B–15B) average 90.8%, indicating that vulnerability-type encoding is consistent across the tested scales (6.7B–15B). We do not claim scale invariance beyond this range: whether substantially

larger models reorganize this signal remains an open question.

The encoding asymmetry. The model encodes distinct CWE-type patterns (buffer overflows, injection attacks, path traversal, resource leaks) with median pairwise AUROC 0.877. Vulnerable-vs.-secure classification achieves only peak AUROC 0.648 with mean-token pooling, using the same aggregation, probe, representation, and language. The +0.23 AUROC gap reveals that CWE-type patterns show higher linear separability, while vulnerability status (vulnerable/secure) shows lower categorical accuracy under these settings.

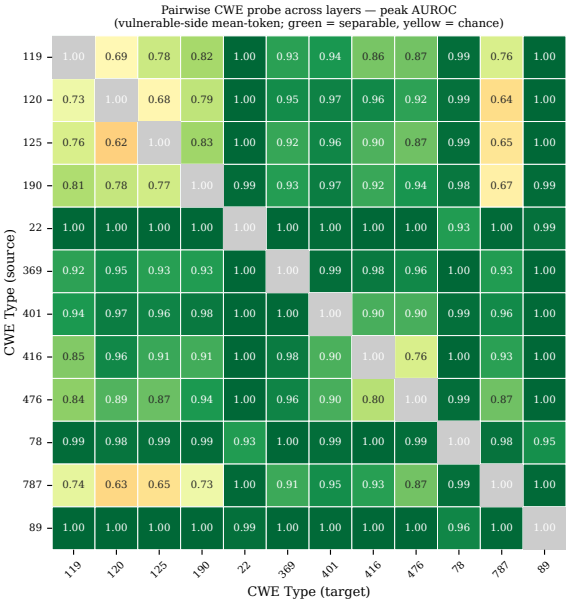


Figure 1: **Pairwise CWE-type probe AUROC.** Heatmap for Qwen2.5-7B-Instruct on DELTASECOMMITS (vulnerable-side mean-token pooling, Layer 19). The 12 CWE types include memory violations (CWE-119, CWE-120, CWE-125, CWE-787), injection attacks (CWE-78, CWE-89), path traversal (CWE-22), and resource leaks (CWE-401). Most pairs are highly separable (90–100% AUROC, dark green), with weaker separation between similar memory-related types (58–72% AUROC, yellow/orange). Color scale: green indicates strong separation, yellow indicates chance-level performance. This pattern reflects strong functional encoding of vulnerability categories across diverse CWE types.

5 How Vulnerability Is Encoded

Section 4 showed the model achieves high fidelity encoding *what code does*, but achieves limited categorical accuracy on *whether it is safe*. We now ask: where in the model's representations does this safety signal live? We find a consistent pattern across 8 code models spanning three model families and two size categories and three vulnerability datasets: the vulnerability signal exists and is geometrically stable, and—under our paired C-code datasets and linear-probing setup—is more accessible as a paired relational signal than as a globally separable categorical label.

5.1 Consistency

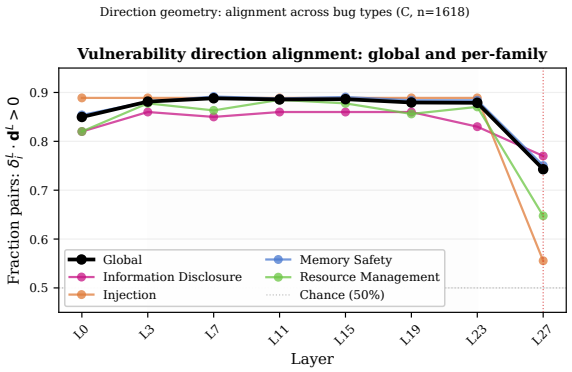


Figure 2: Vulnerability direction alignment across layers and bug types. Global alignment (black, bold) reaches 87–88% across L3–L23, dropping to 70% at L27. Per-family breakdown (colored lines) shows consistent alignment across bug types: Memory Safety (blue), Injection (orange), Resource Management (green), Information Disclosure (magenta). Family-level alignment closely tracks global alignment across all bug types.

Multi-model Vulnerability direction consistency We find a stable, high-alignment vulnerability direction—replicates across all 8 models and all three vulnerability datasets. SVEN produces alignment curves (86–87% peak, 70% at L27) and PRECISEBUGS shows the same geometry. All three datasets show alignment profiles within 2 percentage points across all three models. Cross-model consistency is high (mean Pearson $r = 0.921$ across the three model pairs), indicating that per-pair alignment values are nearly identical across the three independently trained architectures.

(1) Geometric stability across layers. The vulnerability direction is geometrically stable: cosine similarity between directions at different layers reaches 0.988–0.996 for all L3–L23 pairs (mean: 0.993, $n = 15$ pairwise comparisons). The direction remains essentially identical until the final layer L27 (cosine similarity drops to ≈ 0).

(2) High per-pair consistency. For 87–88% of all vulnerable–secure pairs, the vulnerable representation lies further along the vulnerability direc-

tion than its secure counterpart (vs. 50% baseline for random alignment). This alignment is stable from L3 to L23, dropping to 70.19% at L27, reflecting the AUROC collapse at the final layer. Stability persists within L3–L23—showing vulnerability geometry emerges early and persists until the final layer, with 87–88% representing a 74% relative improvement over chance.

(3) Cross-family transfer. We compute separate directions for memory-safety, injection, resource leaks, and information disclosure, then test whether directions trained on one family align with pairs from other families (Figure 4). A direction learned from buffer-overflow family aligns with 83.9% of information-disclosure pairs and 80.0% of resource-leak pairs (vs. 50% chance). Mean transfer across all families: 83.9%, with memory-safety and information-disclosure at 81–87%. Injection attacks show weaker cross-family alignment (70–75%). This cross-family transfer indicates that directions are not entirely family-specific.

Generalization across model scales and families. The relational signal is consistent across the tested scales (6.7B–15B). Mean per-pair alignment (L3–L23) is nearly identical across model sizes: 7B models (Qwen2.5-7B-Instruct, CodeLlama-7B-Instruct, StarCoder2-7B) at 85.85%, 13B–15B models (Qwen2.5-14B-Instruct, CodeLlama-13B-Instruct, StarCoder2-15B) at 85.58%, and alternative 7B models (DeepSeek-Coder-6.7B-Instruct, Qwen2.5-Coder-7B-Instruct) at 83.23%.

5.2 Relational vs. categorical encoding

Why does this signal not support categorical classification with linear probes? The direction captures *paired comparisons*—in 87–88% of pairs, the vulnerable code projects further along $\hat{d}$ than the secure code. However, in the full population, vulnerable and secure samples form overlapping distributions in projection space. This means no linear threshold on the identified vulnerability direction can separate them globally. The vulnerability information is encoded as relational structure—relative orderings between code pairs—rather than as categorical clusters, which limits the effectiveness of standard linear classification in absolute terms.

A representation-level constraint in the tested settings: This overlap persists across all models. Instead of globally separable clusters, the knowledge is encoded as pairwise orderings—"code A is more vulnerable than code B"—which is structurally distinct from absolute categorical labels—"code X is vulnerable."

Under standard linear probing the model achieves only modest absolute classification, but reliably ranks which code is riskier. This makes the model well-suited for prioritization and code-review triage, where relative risk matters more than absolute thresholds.

What each metric does and does not imply. The two metrics answer different questions. Per-pair alignment is a *relative* measure: given a matched (vulnerable, secure) pair, it reports whether the vulnerable version projects further along $\hat{d}$ than its secure counterpart. It presupposes a paired reference and quantifies ordering *within* that reference. The binary probe is an *absolute* measure: it asks whether a single, isolated representation can be labeled vulnerable or secure with no counterpart available. High per-pair alignment (87–88%) therefore establishes that the direction encodes relative risk, but it does *not* imply deployable vulnerability detection. In a realistic setting the model sees isolated code with no matched-secure reference, and absolute classification on the same direction is near chance (mean-token AUROC ≈ 0.648); an explicit unpaired classification setting confirms this directly (peak AUROC 0.634–0.648; Appendix C). Our alignment result should be read as evidence about representational structure, not as a detection score. Converting it into deployable detection requires supplying the missing reference—either a paired counterpart or a calibrated comparison set against which a sample's projection can be scored. Code review offers a natural instance of such a reference: a proposed change supplies its own before/after pair, so the direction can be applied as a lightweight helper process that flags whether an edit moves the representation toward the safer or the riskier side. In this pre-/post-change setting the relational signal is directly actionable—ranking or gating changes by the sign and magnitude of their projection shift—without requiring the absolute, per-sample classification that our linear probes do not recover.

Ruling out length artifacts. Vulnerable code is 20% longer on average in DELTASECOMMITS, raising the question: could the vulnerability direction reflect sequence length rather than vulnerability properties? To test whether length is a confound, we employ a residualization approach: for each

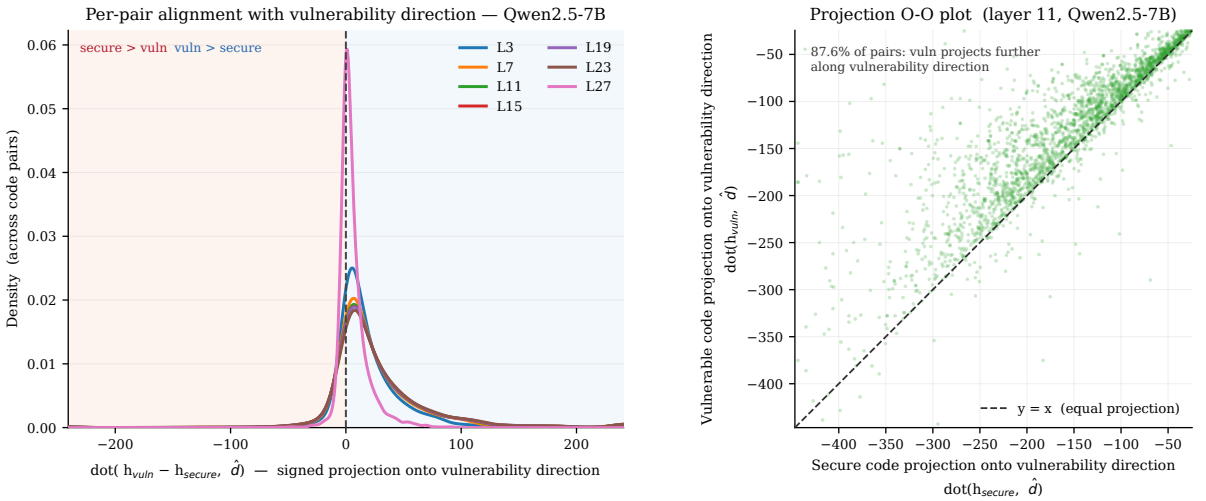


Figure 3: Projection distributions (Qwen2.5-7B-Instruct, mean-token pooling). **Left**: Distribution of pair-wise projection differences $(h_{\text{vuln}} - h_{\text{secure}}) \cdot \hat{d}$ across layers. Most pairs exhibit small positive differences concentrated near zero, with $\sim$13% negative. **Right**: Vulnerable versus secure projections. The tight clustering around the diagonal and strong overlap between distributions indicate modest per-pair separation and no global linear decision boundary.

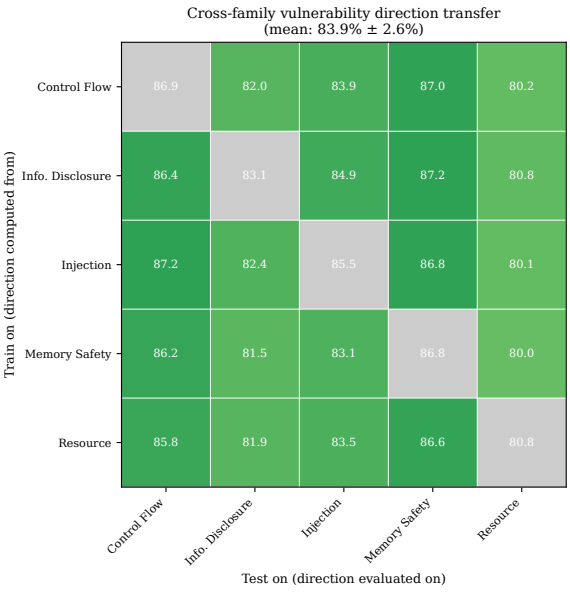


Figure 4: Cross-family vulnerability direction transfer heatmap (Qwen2.5-7B-Instruct, DELTASECOMMITS C subset, aggregated across layers L0–L27). Each cell shows per-pair alignment when a direction trained on the row family (on training data) is tested on the column family (on held-out test data), computed per-fold to maintain train/test separation. Values represent mean transfer across all layers. Warm colors (82–87%) indicate strong transfer; cooler colors (70–75%) indicate weaker transfer for injection attacks. The diagonal is undefined (comparing direction to itself). Mean cross-family transfer: 83.9%, well above chance (50%).

layer, we fit a regression model (Ridge with cross-validation) to predict activations from token count separately within vulnerable and secure samples. We then recompute the vulnerability direction on the residualized activations (actual minus length-predicted component). If the direction persists after length residualization, length differences do not fully explain the direction. This control is important because learned models can absorb length correlations; showing the direction survives length adjustment indicates the signal is not entirely length-driven. A per-model, per-layer breakdown confirms this: after regressing out log character count, the peak binary-probe AUROC remains 0.64–0.66 across all 8 models, essentially unchanged from the uncontrolled peak of 0.648 (Table 1).

6 Discussion

Implications for vulnerability detection. Across all 8 models studied, and under the paired C-code datasets and linear-probing setup we consider, vulnerability information is more accessible as a paired relational signal than as a globally separable categorical label. Representation-based categorical classification via linear probes on standard pooling is insufficient for standalone deployment; ranking-based and preference-learning approaches align with how vulnerability is encoded in the models tested. Recent work on constrained decoding (Fu et al., 2024) and secure code generation (Zhang et al., 2024) demonstrates

Table 1: **Length-controlled binary.** Vulnerable vs. secure probe AUROC with 95% bootstrap CIs across layers, for all 8 models. Code length is regressed out before mean-token probing. Bold marks each model's peak layer.

Model	L3	L7	L11	L15	L19	L23	L27
CodeLlama-7B-Instruct	0.610 [0.592, 0.628]	0.626 [0.609, 0.644]	**0.644** [0.626, 0.659]	0.640 [0.623, 0.656]	0.626 [0.608, 0.643]	0.598 [0.580, 0.615]	0.578 [0.558, 0.594]
CodeLlama-13B-Instruct	0.611 [0.592, 0.628]	0.640 [0.623, 0.656]	**0.646** [0.629, 0.663]	0.644 [0.627, 0.661]	0.634 [0.616, 0.650]	0.618 [0.600, 0.635]	0.590 [0.572, 0.608]
DeepSeek-Coder-6.7B-Instruct	0.610 [0.591, 0.628]	0.631 [0.614, 0.648]	**0.643** [0.625, 0.659]	0.635 [0.618, 0.653]	0.630 [0.612, 0.647]	0.614 [0.595, 0.631]	0.607 [0.590, 0.624]
Qwen2.5-7B-Instruct	0.613 [0.595, 0.632]	0.629 [0.611, 0.646]	0.628 [0.610, 0.645]	0.645 [0.626, 0.661]	**0.654** [0.637, 0.670]	0.623 [0.605, 0.640]	0.604 [0.585, 0.622]
Qwen2.5-Coder-7B-Instruct	0.603 [0.584, 0.622]	0.627 [0.609, 0.644]	0.620 [0.602, 0.637]	0.648 [0.629, 0.665]	**0.657** [0.640, 0.674]	0.614 [0.595, 0.631]	0.585 [0.566, 0.603]
Qwen2.5-14B-Instruct	0.610 [0.593, 0.629]	0.625 [0.607, 0.643]	0.608 [0.589, 0.626]	0.620 [0.602, 0.637]	0.631 [0.614, 0.648]	0.643 [0.625, 0.659]	**0.646** [0.628, 0.662]
StarCoder2-7B	0.625 [0.607, 0.643]	**0.644** [0.625, 0.661]	0.637 [0.620, 0.654]	0.634 [0.618, 0.652]	0.628 [0.611, 0.645]	0.610 [0.593, 0.627]	0.596 [0.578, 0.613]
StarCoder2-15B	0.617 [0.599, 0.636]	**0.641** [0.623, 0.658]	0.632 [0.615, 0.649]	0.624 [0.607, 0.641]	0.634 [0.616, 0.650]	0.631 [0.613, 0.648]	0.606 [0.588, 0.623]

the effectiveness of methods that leverage ordinal relationships in code representations.

Implications for representation structure in code generation. Our conclusion is not that absolute vulnerability classification is impossible, but that it is weakly recoverable under our linear probing setup. A shallow MLP probe on the same representations (64 hidden units; Appendix B) yields peak AUROC 0.635–0.644 across models, essentially unchanged from the linear-probe peak of 0.648, indicating that a lightweight nonlinear readout does not recover materially more categorical signal. We do not preclude that more expressive readouts recover stronger absolute performance—larger nonlinear probes, learned or token-level pooling, contrastive objectives, or prompt-based classifiers with richer natural-language context. Rather, our findings characterize what is accessible with linear and lightweight nonlinear readouts on standard pooling: the model reliably ranks which code is riskier, but categorical classification achieves limited accuracy due to overlapping distributions. Whether more expressive nonlinear methods can recover higher categorical accuracy on this representation remains an open question orthogonal to characterizing the representation's structure. Our analysis identifies a single global vulnerability direction with strong geometric properties: stability across layers, consistency across 8 model architectures and scales, and transferability across bug families. These consistent properties demonstrate a unified relational vulnerability signal in these models. However, whether a single global direction suffices for all vulnerability types and contexts, or whether mixtures/local directions could better capture context-dependent vulnerability semantics, remains an empirical question. Recent representation-engineering work such as Yu et al. (2025) (mixtures of control vectors), Huang et al. (2026), and Yousefpour et al. (2025) has explored leveraging learned directions to influence model behavior. Systematic evaluation of whether single vs. mixture directional approaches are most effective for steering on the identified vulnerability direction is future work. Related approaches like Adila et al. (2026) and Belitsky et al. (2025) propose complementary mechanisms (weight updates, KV cache interventions) for controlled manipulation.

Limitations and scaling implications. Our conclusions are scoped to the studied setting and should not be read as claims about code models or LLMs in general. (1) Model coverage spans 8 open models in the 6.7B–15B range; whether the same representation geometry holds for substantially larger or closed frontier models is untested. (2) The analysis is intentionally restricted to C programs to control for language-specific variation while isolating representational properties, so transfer to other programming languages (Python, Java, JavaScript) is unverified; evaluating additional languages is an important direction for future work. (3) We probe fixed, context-free representations; instruction-following vulnerability detection with richer natural-language context is outside our scope and may expose additional categorical structure. (4) Reported AUROC values depend critically on two factors. Cross-validation methodology (pair-stratified vs. standard fold; see Methodology 3) contributes +0.055 for last-token, +0.016 for mean-token. Token aggregation strategy contributes +0.117. (5) Paired vulnerable/patched datasets naturally encode a before/after relationship, so the learned direction may capture general code-change signals in addition to vulnerability-specific patterns. Disentangling the two would require comparable paired data from other software-engineering activities such as refactoring or code simplification. Such data remain scarce for C: Ritz et al. (2025) introduce a C++ refactoring dataset with only 16 examples, and building reliable resources would require mining commits explicitly marked as refactoring together with robust valida-

tion. We leave this investigation to future work.

Reproducibility and hyperparameter details. To support future replication and mechanistic interpretability work, we intend to release all code, activation caches, and analysis scripts upon publication. **Probing:** Linear logistic regression, L2 regularization ($\lambda = 1.0$), balanced class weighting. PCA-50 projection (61.3% variance), fitted independently per fold. PCA-100 differs <0.01 (Appendix A). **Direction computation:** Vulnerability direction $\mathbf{d} = \text{normalize}(\bar{v} - \bar{s})$ per layer, using mean activations from training data.

Advances in the field have hindered the practical adoption of LLMs in software vulnerability detection, including limited robustness in code representations, vulnerability data leakage, and limited interpretability (Kaniewski et al., 2025; Mao et al., 2024; Sheng et al., 2025; Zheng et al., 2023; Ramírez et al., 2024). We hope our findings on the representational structure of vulnerability signals can inform future model design and future training strategies to address these limitations, particularly through comprehensive approaches (Dai et al., 2025) that account for the greater accessibility of vulnerability information as a relational signal than as a globally separable categorical label in these settings.

The consistent direction vectors across layers align with recent work on representation engineering (Zou et al. (2023)), particularly approaches like Yu et al. (2025) which leverage collections of linear directions to steer code generation. Recent work on contrastive learning and preference modeling are naturally aligned with relational vulnerability signals, given that these methods directly optimize pairwise comparisons. Empirical validation of steering and preference-learning approaches on this direction remains future work. Complementary approaches such as integrated static analysis and fuzz testing (Li et al., 2024), combined with knowledge-level retrieval-augmented generation (Du et al., 2024), represent strategies for enhancing LLM-based vulnerability detection by combining relational signals with additional safety mechanisms. Broader strategies including graph-based vulnerability detection (Lu et al., 2024), dataset-driven vulnerability mitigation (Wang et al., 2023), similarity-based embeddings (Shimmi et al., 2024), and systematic evaluation frameworks (Peng et al., 2025) demonstrate the diversity of approaches complementary to relational encoding.

Our work is a representation analysis, not an end-to-end system. Recent work shows that richer context and structured reasoning substantially improve absolute detection. Those findings are orthogonal to and consistent with ours, since we characterize how vulnerability is organized in fixed, context-free representations. The relative encoding we observe is best exploited by ranking-based retrieval and human-in-the-loop triage—comparing the relative risk of code samples rather than performing direct binary classification. Detection systems could combine this relational signal with additional context and reasoning mechanisms.

7 Conclusion

The code models we study understand vulnerability, but encode it in a form that differs from how we typically deploy them. They reliably rank pairs of code snippets by risk (87–88% per-pair alignment) yet achieve only moderate categorical classification accuracy (peak AUROC 0.648). Under the studied paired C-code datasets and linear probing, the vulnerability signal is more accessible as relational structure—comparative judgments like "this is worse than that"—than as a globally separable categorical label.

Disclosure

AI was used to improve the clarity and organization of this manuscript.

Acknowledgments

Rui Melo (2024.18549.PRT) is funded by Fundação para a Ciência e Tecnologia (FCT) This work is also supported by the SALVE project (2024.14936.PEX) and HPC (2025.08039.CPCA).

Societal Impact

Our work is motivated by a defensive objective: understanding how code LLMs internally represent vulnerability-related information to support human-in-the-loop vulnerability triage and safer code-generation systems. We acknowledge a dual-use risk: a stable vulnerability direction could reduce the cost of prioritizing code for security inspection. Although our techniques build on existing representation analysis methods, they may lower the barrier to misuse. We therefore present this work as a resource for defensive evaluation and understanding model behavior, rather than as an operational attack toolkit.

References

Dyah Adila, John Cooper, Alexander Yun, Avi Trost, and Frederic Sala. 2026. Weight updates as activation shifts: A principled framework for steering. *Preprint*, arXiv:2603.00425.

Guillaume Alain and Yoshua Bengio. 2017. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*.

Anthropic. 2026. Project glasswing: Partnering on ai-assisted security. Web resource.

Anthropic and 1 others. 2026. Claude mythos preview: Cybersecurity capabilities assessment. *Anthropic Red Team*.

Max Belitsky, Dawid J. Kopiczko, Michael Dorkenwald, M. Jehanzeb Mirza, James R. Glass, Cees G. M. Snoek, and Yuki M. Asano. 2025. Kv cache steering for controlling frozen llms. *Preprint*, arXiv:2507.08799.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345.

Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the 45th International Conference on Software Engineering*, pages 121–133. IEEE.

Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2025. A comprehensive study of llm secure code generation. *arXiv preprint arXiv:2503.15554*.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability detection with code language models: How far are we? In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM.

Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, and 1 others. 2024. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*.

Mohamed Amine Ferrag, Ammar Battah, Norbert Tihanyi, Ridhi Jain, Diana Maimuţ, Fatima Alwahedi, Thierry Lestable, Narinderjit Singh Thandi, Abdechakour Mechri, Merouane Debbah, and Lucas C. Cordeiro. 2025. Securefalcon: Are we there yet in automated software vulnerability detection with llms? *IEEE Transactions on Software Engineering*.

Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636. IEEE.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218*.

Zhaolin Gao, Jonathan D Chang, Wenhao Zhan, Owen Oertell, Gokul Swamy, Kianté Brantley, Thorsten Joachims, J Andrew Bagnell, Jason D Lee, and Wen Sun. 2024. Rebel: Reinforcement learning via regressing relative rewards. *Advances in Neural Information Processing Systems*, 37:52354–52400.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879.

Y. He, Z. Chen, and C. Le Goues. 2023. Precisebug-collector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1899–1910, Los Alamitos, CA, USA. IEEE Computer Society.

John Hewitt and Percy Liang. 2019a. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 2733–2743.

John Hewitt and Percy Liang. 2019b. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743, Hong Kong, China. Association for Computational Linguistics.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4129–4138.

Li Huang, Meng Yan, Tao Yin, Weifeng Sun, Zhongxin Liu, Hongyu Zhang, and David Lo. 2026. Steer your model: Secure code generation with contrastive decoding. *IEEE Transactions on Software Engineering*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang,

Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Dipanjan Das Ian Tenney and Ellie Pavlick. 2019. Bert rediscovers the classical nlp pipeline. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Sabrina Kaniewski, Fabian Schmidt, Markus Enzweiler, Michael Menth, and Tobias Heer. 2025. A systematic literature review on detecting software vulnerabilities with large language models. *ArXiv*, abs/2507.22659.

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169*.

Haonan Li, Yu Hao, Yizhuo, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards explainable vulnerability detection with large language models. *IEEE Transactions on Software Engineering*, 51:2957–2971.

Rui Melo, Sofia Reis, Andre Catarino, and Rui Abreu. 2026. Do language models prefer vulnerable code? a probabilistic study of insecure code preference. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in GPT. In *Advances in Neural Information Processing Systems*, volume 35.

Frederick Mosteller. 1951. Remarks on the method of paired comparisons: Iii. a test of significance for paired comparisons when equal standard deviations and equal correlations are assumed. *Psychometrika*, 16(2):207–218.

Kiho Park, Yo Joong Choe, and Victor Veitch. 2024. The linear representation hypothesis and the geometry of large language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 33–40. IEEE.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741.

Leonardo Criollo Ramírez, Xavier Limón, Ángel J Sánchez-García, and Juan Carlos Pérez-Arriaga. 2024. State of the art of the security of code generated by llms: A systematic literature review. In *2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 331–339. IEEE.

Francisco Ribeiro, Claudio Spiess, Prem Devanbu, and Sarah Nadi. 2026. On llms' internal representation of code correctness. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*.

B. Ritz, A. Karakaš, and D. Helic. 2025. Refactoring detection in C++ programs with RefactoringMiner++. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1163–1167.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. Llms in software security: A survey of vulnerability detection techniques and insights. *Preprint*, arXiv:2502.07049.

Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. 2024. {VulSim}: Leveraging similarity of {Multi-Dimensional} neighbor embeddings for vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1777–1794.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601.

Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496.

Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880. IEEE.

Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2023. Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation. *arXiv preprint arXiv:2310.16263*.

Ashkan Yousefpour, Taeheon Kim, Ryan Sungmo Kwon, Seungbeen Lee, Wonje Jeung, Seungju Han, Alvin Wan, Harrison Ngan, Youngjae Yu, and Jonghyun Choi. 2025. Representation bending for large language model safety. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24073–24098.

Weichen Yu, Ravi Mangal, Terry Zhuo, Matt Fredrikson, and Corina S. Pasareanu. 2025. A mixture of linear corrections generates secure code. *arXiv preprint arXiv:2507.09508*.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. Seccoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

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, and 2 others. 2023. Representation engineering: A top-down approach to AI transparency. *arXiv preprint arXiv:2310.01405*.

Appendix

A PCA Dimensionality Sensitivity

Question. *Do the mean-token probing results depend on the choice of 50 PCA components, or are they stable across dimensionalities?*

We test robustness of mean-token probing across PCA dimensionalities (10, 20, 50, 100, 200 components). All other settings match the main paper (5-fold stratified CV, Ridge logistic regression). We use Qwen2.5-7B-Instruct and DELTASECOMMITS C subset for this analysis. Table 1 and Figure 1 show mean-token AUROC across layers and PCA dimensionalities.

Table 1: **PCA sensitivity.** Mean-token AUROC across layers and PCA dimensionalities. Paper setting is PCA-50.

Layer	PCA-10	PCA-20	PCA-50*	PCA-100	PCA-200
L0	0.578	0.593	0.600	0.592	0.575
L3	0.578	0.597	0.614	0.616	0.591
L7	0.576	0.599	0.624	0.625	0.615
L11	0.587	0.602	**0.633**	0.632	0.621
L15	0.602	0.638	**0.647**	0.645	0.632
L19	0.584	0.619	0.635	0.637	0.634
L23	0.555	0.567	0.596	0.596	0.590
L27	0.485	0.550	0.585	0.578	0.561

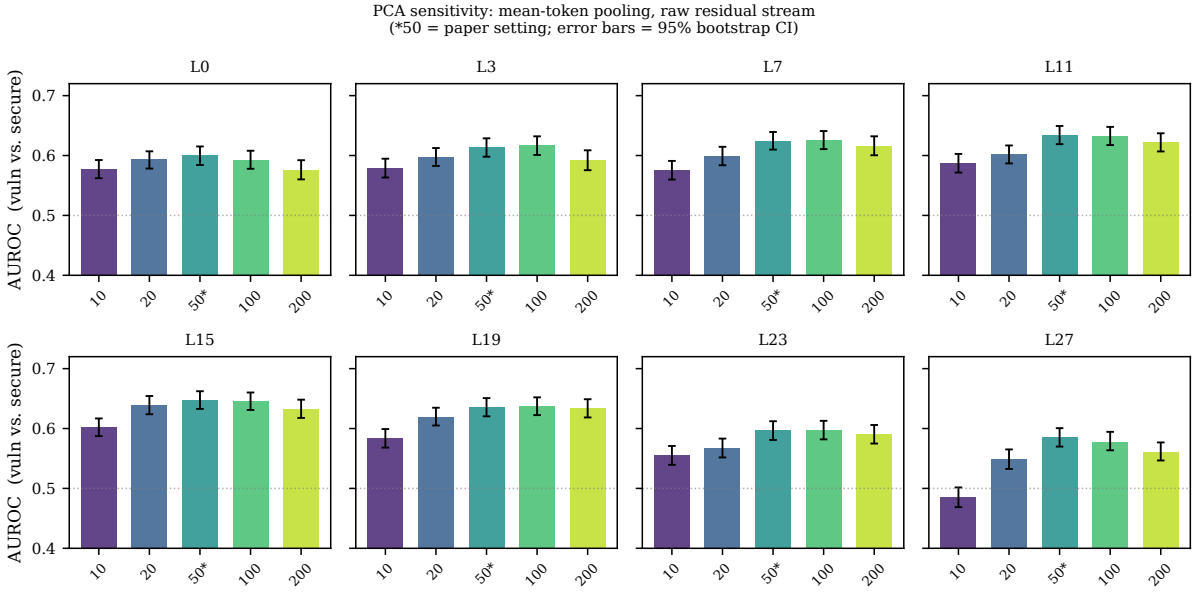


Figure 1: **PCA sensitivity.** Mean-token AUROC at each of the eight layers under five PCA dimensionalities.

Key findings: (1) PCA-50 is near-optimal. PCA-100 identical within CI, PCA-20 recovers most signal. (2) Higher dimensionality does not improve results.

Finding. PCA-50 and PCA-100 agree to within 0.010 AUROC at every layer. PCA-20 recovers most of the signal.

B Nonlinear (MLP) Probe

Our main results use linear probes as an interpretability diagnostic: they characterize what is *linearly* accessible without conflating the representation's structure with a probe's own capacity to memorize or combine weak signals (Hewitt and Liang, 2019a). To test whether a modestly more expressive readout recovers additional categorical signal, we replace the linear probe with a shallow MLP (64 hidden units, ReLU activation, Adam optimizer, learning rate 0.001) on the same mean-token representations, all 8 models, and identical 5-fold stratified cross-validation.

Table 2 reports per-model, per-layer AUROC with 95% bootstrap CIs. Peak AUROC across models ranges 0.635–0.644, essentially unchanged from the linear-probe peak of 0.648.

Table 2: **Nonlinear (MLP) probe.** Vulnerable vs. secure classification AUROC with 95% bootstrap CIs across layers, for all 8 models. A shallow MLP (64 hidden units, ReLU, Adam, lr$= 0.001$) replaces the linear readout on the same mean-token representations. Bold marks each model's peak layer. Peak AUROC (0.635–0.644) is essentially unchanged from the linear-probe peak (0.648), indicating that a lightweight nonlinear readout does not recover materially more categorical signal.

Model	L3	L7	L11	L15	L19	L23	L27
CodeLlama-7B-Instruct	0.610 [0.593, 0.628]	0.621 [0.603, 0.638]	**0.635** [0.617, 0.652]	0.630 [0.611, 0.648]	0.617 [0.599, 0.635]	0.600 [0.582, 0.618]	0.601 [0.582, 0.618]
CodeLlama-13B-Instruct	0.610 [0.593, 0.627]	**0.637** [0.620, 0.654]	0.635 [0.617, 0.651]	0.636 [0.619, 0.652]	0.626 [0.610, 0.643]	0.610 [0.592, 0.626]	0.606 [0.588, 0.624]
DeepSeek-Coder-6.7B-Instruct	0.605 [0.587, 0.623]	0.632 [0.615, 0.649]	**0.639** [0.622, 0.656]	0.628 [0.610, 0.645]	0.618 [0.600, 0.636]	0.609 [0.590, 0.626]	0.603 [0.585, 0.620]
Qwen2.5-7B-Instruct	0.611 [0.594, 0.629]	0.634 [0.617, 0.652]	0.633 [0.616, 0.650]	**0.644** [0.626, 0.662]	0.641 [0.625, 0.658]	0.619 [0.601, 0.636]	0.604 [0.587, 0.623]
Qwen2.5-Coder-7B-Instruct	0.606 [0.588, 0.624]	0.632 [0.614, 0.649]	0.617 [0.600, 0.635]	**0.644** [0.627, 0.661]	0.642 [0.626, 0.658]	0.611 [0.593, 0.629]	0.599 [0.581, 0.616]
Qwen2.5-14B-Instruct	0.599 [0.581, 0.616]	0.620 [0.602, 0.637]	0.608 [0.590, 0.623]	0.625 [0.608, 0.642]	0.633 [0.615, 0.651]	**0.642** [0.624, 0.658]	0.639 [0.622, 0.657]
StarCoder2-7B	0.628 [0.610, 0.645]	**0.638** [0.622, 0.656]	0.632 [0.616, 0.649]	0.628 [0.610, 0.647]	0.625 [0.608, 0.642]	0.613 [0.595, 0.629]	0.602 [0.584, 0.620]
StarCoder2-15B	0.621 [0.605, 0.638]	**0.636** [0.619, 0.654]	0.634 [0.615, 0.649]	0.629 [0.612, 0.646]	0.632 [0.614, 0.649]	0.633 [0.614, 0.650]	0.605 [0.587, 0.622]

C Unpaired Binary Classification

Our main binary-probe results are trained and evaluated on paired vulnerable/secure samples, which isolates the representational effect of a vulnerability by controlling for functionality and implementation style, but does not by itself represent the practical deployment setting where arbitrary, unmatched code must be classified in isolation. To directly test this, we evaluate an unpaired binary classification setting: vulnerable and secure samples are treated as independent examples with no matched-pair structure, using the same linear probe, mean-token pooling, and 5-fold stratified cross-validation as the main results, for all 8 models.

Table 3 reports per-model, per-layer AUROC with 90% bootstrap CIs. Peak AUROC across models ranges 0.634–0.648, closely matching the paired-evaluation peak of 0.648.

Table 3: **Unpaired binary classification.** Vulnerable vs. secure probe AUROC with 90% bootstrap CIs across layers, for all 8 models. Vulnerable and secure samples are treated as independent, unpaired examples (no matched-reference structure), approximating a realistic setting where arbitrary code is classified in isolation. Bold marks each model's peak layer. Peak AUROC (0.634–0.648) is near-identical to the paired-evaluation peak (0.648), indicating that removing the paired structure does not reveal additional linearly accessible signal.

Model	L3	L7	L11	L15	L19	L23	L27
CodeLlama-7B-Instruct	0.605 [0.588, 0.621]	0.621 [0.603, 0.637]	**0.634** [0.617, 0.650]	0.633 [0.616, 0.649]	0.619 [0.601, 0.635]	0.590 [0.573, 0.608]	0.579 [0.561, 0.596]
CodeLlama-13B-Instruct	0.607 [0.589, 0.623]	0.631 [0.614, 0.647]	**0.638** [0.622, 0.654]	0.637 [0.621, 0.653]	0.627 [0.609, 0.643]	0.611 [0.594, 0.627]	0.590 [0.572, 0.607]
DeepSeek-Coder-6.7B-Instruct	0.607 [0.588, 0.624]	0.626 [0.610, 0.643]	**0.636** [0.620, 0.652]	0.629 [0.612, 0.645]	0.623 [0.606, 0.640]	0.609 [0.590, 0.625]	0.605 [0.586, 0.622]
Qwen2.5-7B-Instruct	0.610 [0.592, 0.627]	0.624 [0.606, 0.640]	0.624 [0.606, 0.640]	0.637 [0.619, 0.653]	**0.645** [0.628, 0.661]	0.619 [0.600, 0.635]	0.602 [0.584, 0.619]
Qwen2.5-Coder-7B-Instruct	0.601 [0.582, 0.619]	0.623 [0.605, 0.640]	0.617 [0.600, 0.633]	0.639 [0.621, 0.654]	**0.648** [0.631, 0.664]	0.606 [0.587, 0.622]	0.583 [0.564, 0.601]
Qwen2.5-14B-Instruct	0.606 [0.589, 0.623]	0.619 [0.600, 0.636]	0.604 [0.585, 0.620]	0.617 [0.599, 0.634]	0.627 [0.610, 0.643]	0.634 [0.616, 0.650]	**0.637** [0.619, 0.653]
StarCoder2-7B	0.620 [0.602, 0.637]	**0.635** [0.618, 0.652]	0.632 [0.615, 0.649]	0.630 [0.613, 0.646]	0.624 [0.607, 0.640]	0.606 [0.588, 0.623]	0.593 [0.575, 0.610]
StarCoder2-15B	0.612 [0.594, 0.629]	**0.637** [0.620, 0.653]	0.629 [0.612, 0.645]	0.621 [0.604, 0.638]	0.630 [0.612, 0.646]	0.626 [0.609, 0.643]	0.602 [0.585, 0.619]

D Pair-Stratified Cross-Validation

Question. *Could the vulnerability signal be an artifact of leakage between paired samples during cross-validation?*

Standard 5-fold stratified CV prevents label leakage across folds, but paired datasets can leak if both members of a pair are in different folds. We implemented pair-stratified CV (both members in the same fold) to eliminate within-pair contamination, then compared results using both strategies. We use Qwen2.5-7B-Instruct and DELTASECOMMITS C subset for this analysis.

Isolating CV method vs. pooling strategy. To determine whether the +11% improvement comes from the CV method (pair-stratified vs. standard) or the pooling strategy (mean-token vs. last-token), we present a 2×2 breakdown (Table 5). The pair-stratified CV method contributes marginally (+1–2%) relative to the pooling method, which contributes the dominant effect (+8–16%). This shows that the primary source of improved probing is the use of mean-token pooling, not the validation methodology.

Table 4: Pair-stratified vs. standard cross-validation for binary vulnerability classification (last-token pooling). Both members of each pair are kept in the same fold, eliminating within-pair leakage. Pair-stratified CV shows consistent improvements across all layers. The mean-token counterparts of these values ($0.617 \rightarrow 0.647$) appear in the 2×2 breakdown (Table 5).

Layer	Standard CV	Pair-Stratified CV	Δ AUROC	Δ%
L0	0.445 [0.429–0.460]	0.525 [0.508–0.541]	+0.080	+17.9%
L3	0.540 [0.524–0.555]	0.577 [0.562–0.592]	+0.037	+6.9%
L7	0.496 [0.480–0.511]	0.550 [0.534–0.565]	+0.054	+11.0%
L11	0.497 [0.479–0.512]	0.550 [0.533–0.568]	+0.053	+10.7%
L15	0.514 [0.497–0.530]	0.563 [0.547–0.578]	+0.049	+9.6%
L19	0.523 [0.505–0.540]	0.570 [0.552–0.585]	+0.047	+9.0%
L23	0.497 [0.481–0.511]	0.556 [0.541–0.572]	+0.059	+11.9%
L27	0.489 [0.474–0.505]	0.549 [0.533–0.564]	+0.060	+12.3%
Mean	0.500 [0.481–0.518]	0.555 [0.539–0.570]	+0.055	+11.0%

Table 5: **2×2 breakdown: pooling method vs. CV strategy.** Mean AUROC across 8 layers. Last-token + standard CV establishes the baseline at 0.500. Mean-token pooling is the dominant driver of improvement (+0.117 AUROC), while pair-stratified CV adds a marginal secondary benefit (+0.016 AUROC when combined with mean-token).

	CV Method		
Pooling	Standard	Pair-Stratified	CV Contribution
Last-token	0.500	0.555	+0.055
Mean-token	0.617	0.647	+0.030
Pooling Effect	+0.117	+0.092	

Finding. Pair-stratified CV is not conservative. Rather, it improves results. The 2x2 analysis reveals that most of the reported improvement comes from the pooling method (mean-token vs. last-token, +0.117 AUROC), not the CV strategy. Pair-stratified CV alone contributes +0.055 for last-token and +0.030 for mean-token, whereas mean-token pooling contributes +0.117 under standard CV and +0.092 under pair-stratified CV. This suggests: (1) The main analysis correctly uses mean-token pooling to recover distributed signals; (2) The validation methodology (standard vs. pair-stratified) is a secondary refinement; (3) The vulnerability signal is robust to the stricter split constraint and is not an artifact of pair-member contamination.

E Multi-Model Probing

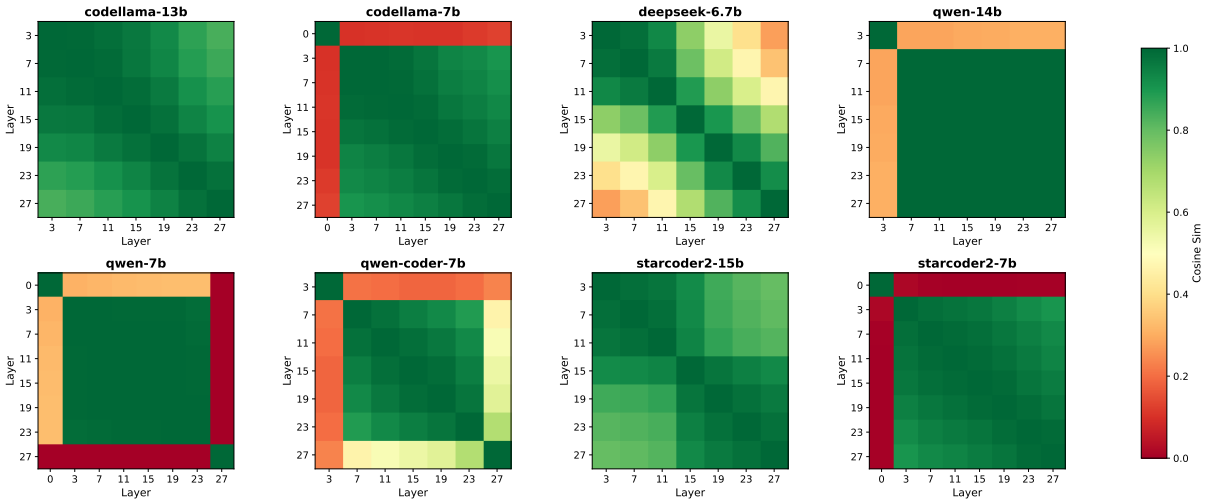


Figure 2: **Cross-layer vulnerability direction alignment.** Three qualitatively distinct patterns emerge. (i) Stable propagation (CodeLlama-13B, StarCoder2-7B, StarCoder2-15B): the direction remains highly consistent across all intermediate and late layers (cosine 0.80-0.99), with mild decay only at the most distant layer pairs. (ii) Progressive rotation (DeepSeek-6.7B): the direction drifts continuously through the network — adjacent layers are tightly aligned (L3-L7 $\sim$ 0.98) but distant pairs diverge substantially (L3-L27 $\sim$ 0.28). (iii) Isolated outlier layers (Qwen-7B, Qwen-Coder-7B, Qwen-14B): intermediate layers form a tight, near-identical block, but one or two layers — typically L3 or L27 — define a direction nearly orthogonal to the rest.

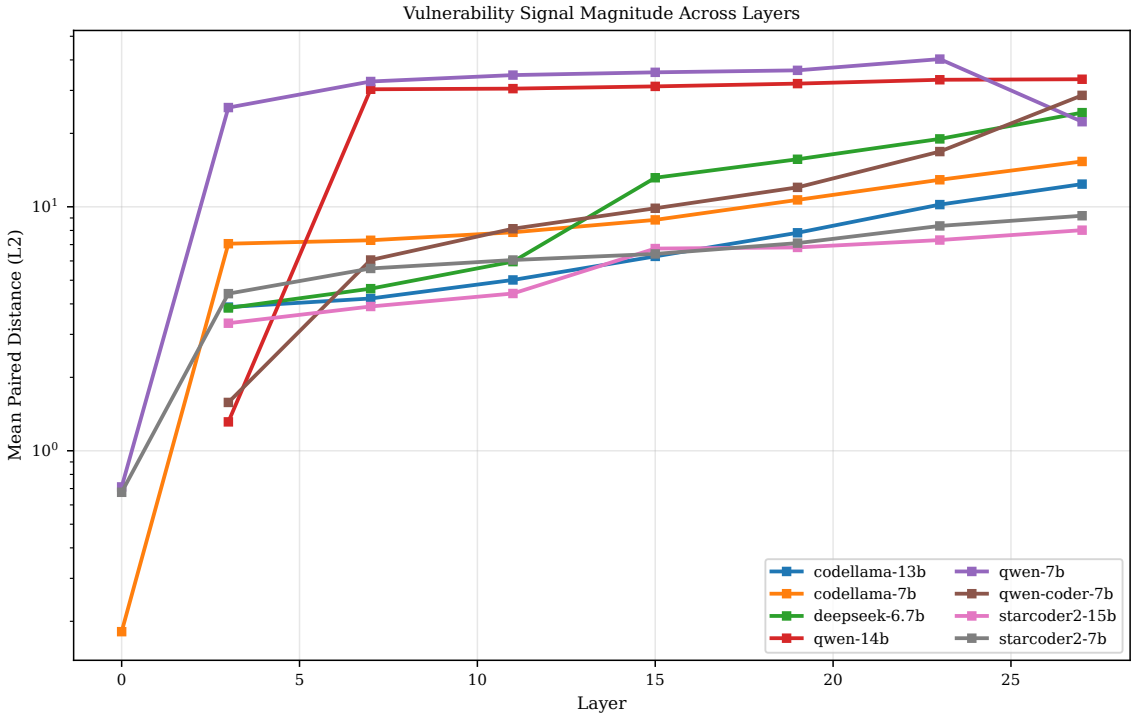


Figure 3: **Mean L2 distance between paired representations (vulnerable vs. secure) across layers for all 8 models.** All models show vulnerability signal magnitude emerging at layer 3 with an $\approx$20–50$\times$ jump from layer 0. The consistent pattern across models indicates that vulnerability information becomes geometrically separable early in the network and persists through mid-layers, supporting the mean-token aggregation strategy.

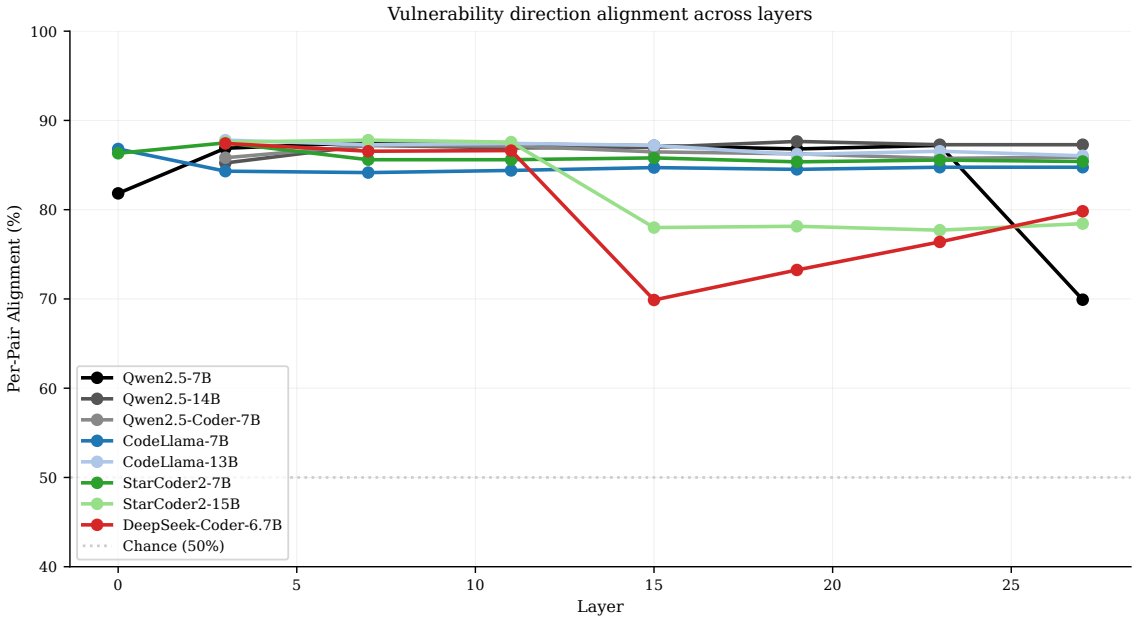


Figure 4: **Vulnerability direction alignment across layers for all models.**

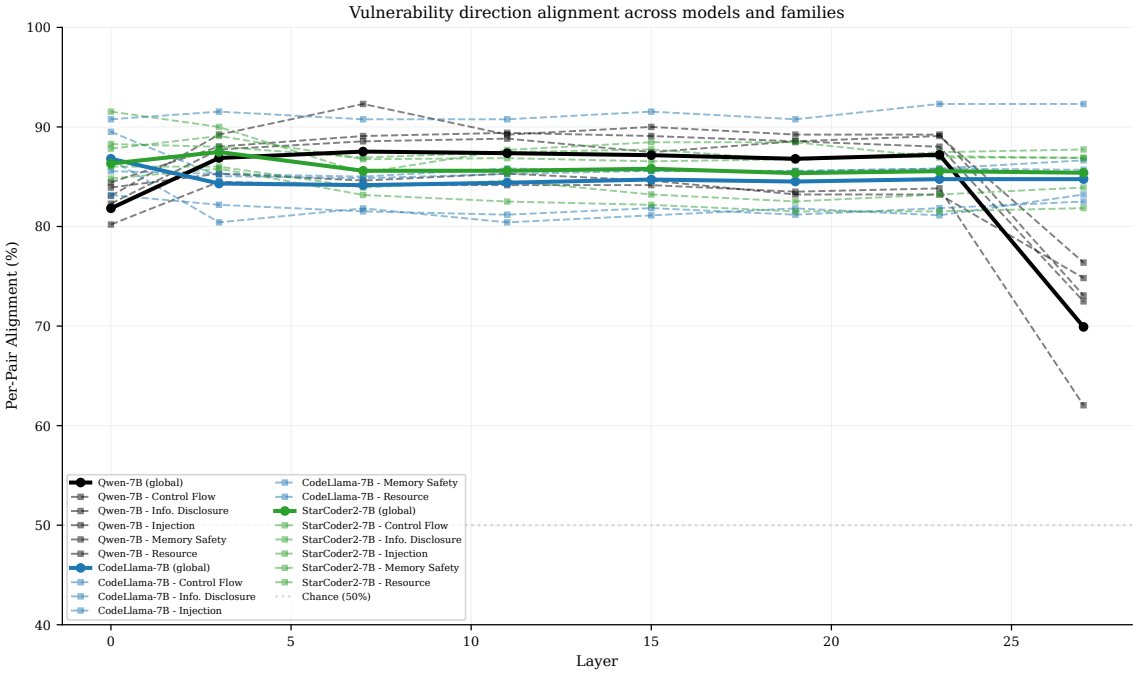


Figure 5: **Vulnerability direction alignment across all models with per-family breakdown on a single plot.** The robustness of per-family alignment confirms that the vulnerability direction captures a general safety property, not model-specific or family-specific patterns.

Direction alignment across layers (per-model comparison).

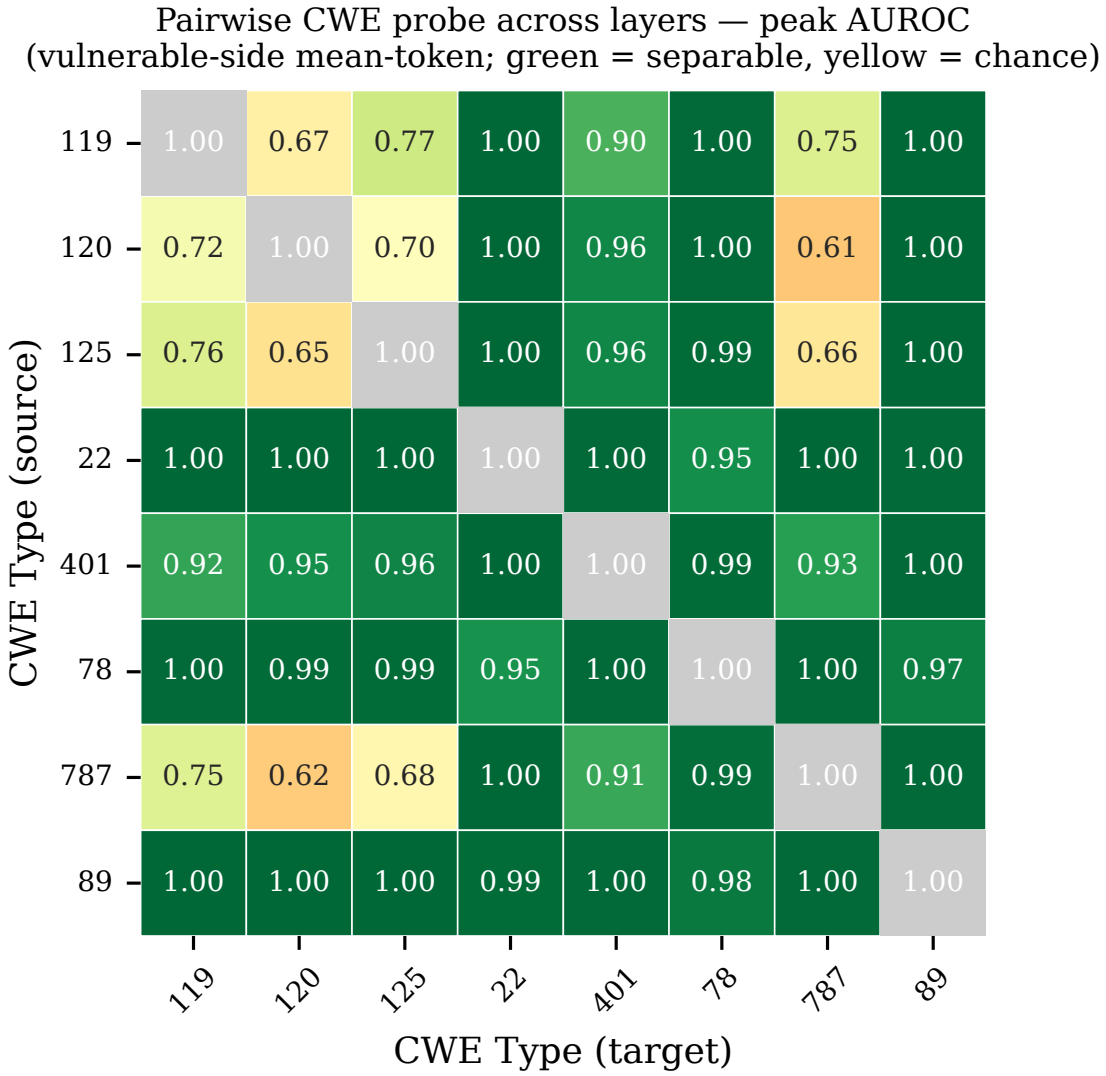


Figure 6: **Pairwise CWE-type probe AUROC for CodeLlama-7B on DELTASECOMMITS (vulnerable-side mean-token pooling, Layer 19)**. The 12 CWE types include CWE-119, CWE-120, CWE-125, CWE-22, CWE-401, CWE-78, CWE-787, CWE-89. Like Qwen-7B, most pairs show strong separation (90–100% AUROC), with weaker separation between memory-related types (CWE-120 vs CWE-125: 74%, CWE-120 vs CWE-78: 59%). Color scale: green indicates high AUROC (separable), yellow indicates chance-level performance. This consistent pattern across models indicates functional CWE understanding is architecture-independent.

Pairwise CWE-type probing across models on DELTASECOMMITS.

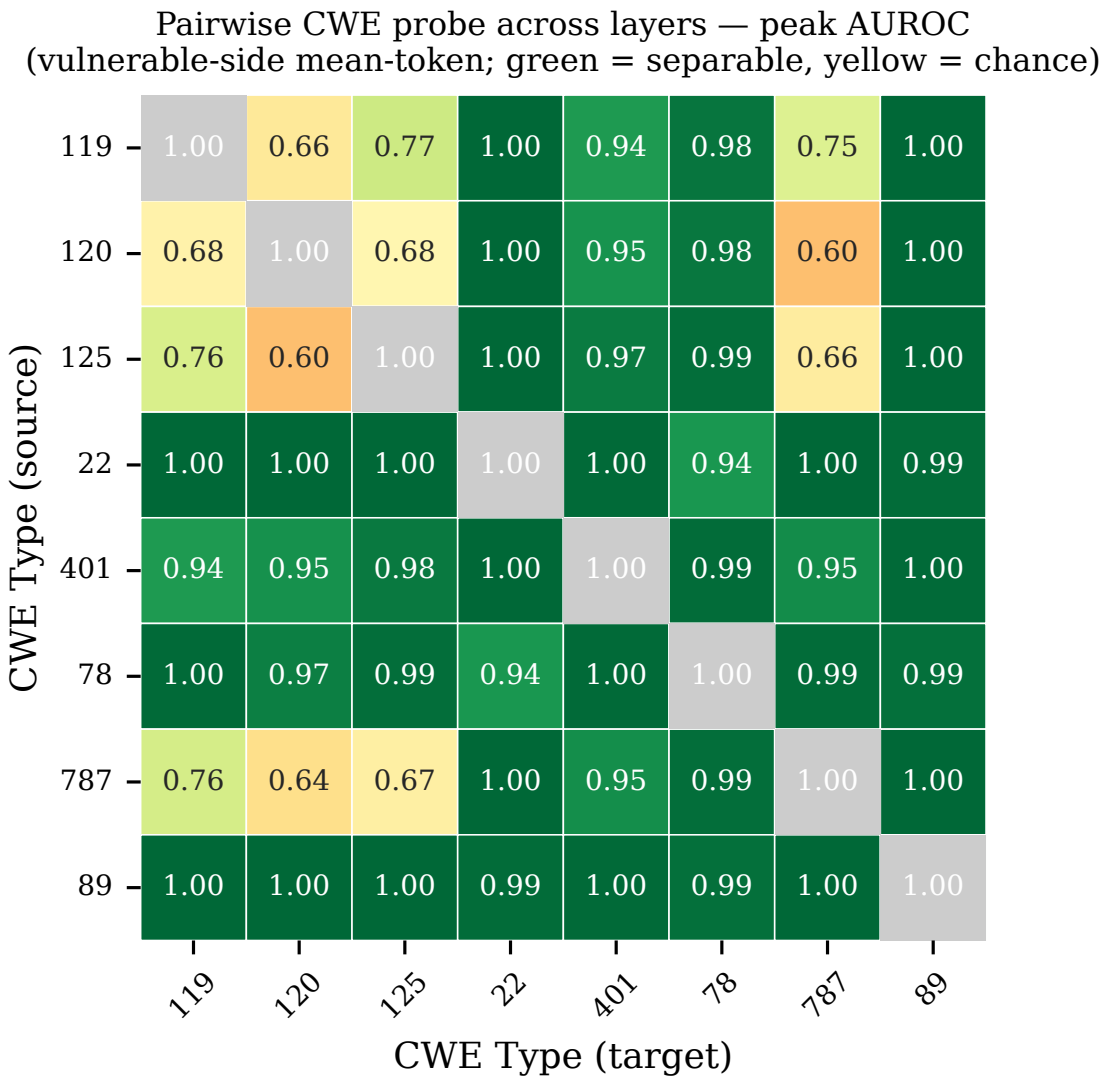


Figure 7: Pairwise CWE-type probe AUROC for StarCoder2-7B on DELTASECOMMITS (vulnerable-side mean-token pooling, Layer 19). The 12 CWE types include CWE-119, CWE-120, CWE-125, CWE-22, CWE-401, CWE-78, CWE-787, CWE-89. StarCoder2 exhibits strong pairwise CWE separation (90–100% AUROC for most pairs), with similar weak-separation pairs as other models. Color scale: green indicates high AUROC (separable), yellow indicates chance-level performance. This demonstrates that functional encoding is consistent across the 7B model family independent of architecture.

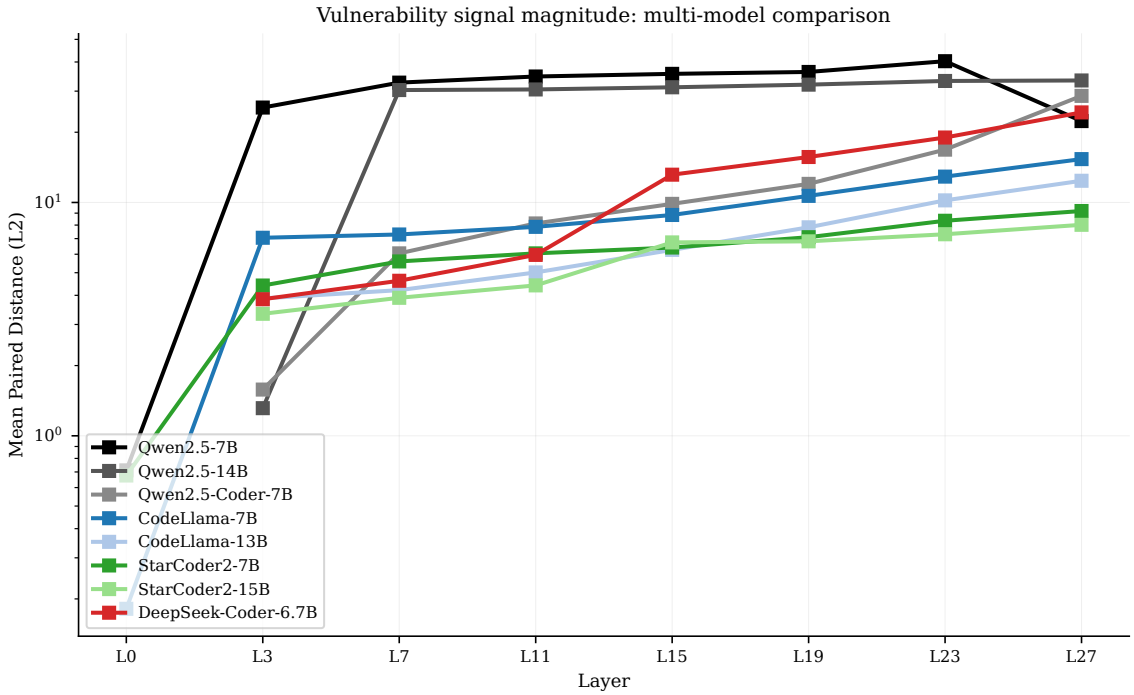


Figure 8: Activation magnitude (mean absolute value) of the vulnerability direction across layers for all three models. Qwen-7B shows the largest magnitude throughout (peaking ≈ 0.5 at L25), while CodeLlama-7B and StarCoder2-7B maintain lower but consistent magnitudes (≈ 0.1–0.2). The parallel growth patterns indicate similar underlying signal structure despite different absolute scales.

Multi-model magnitude and stability.

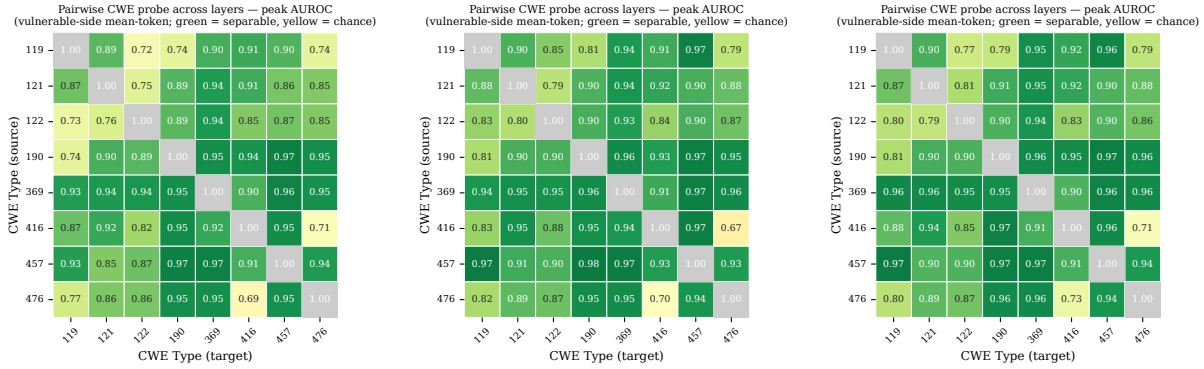
Functional CWE encoding across datasets: PRECISEBUGS. The consistent functional CWE understanding observed in DELTASECOMMITS extends to other vulnerability datasets. We evaluated PRECISEBUGS (real-world C vulnerabilities from open-source repositories, $n = 4{,}101$ pairs, 8 CWE types) using pairwise CWE probing across all three 7B models. PRECISEBUGS achieves 86.9–90.1% mean pairwise AUROC across the three models (Table 6), matching DELTASECOMMITS performance (91.5–91.9%), confirming that functional CWE understanding is robust across different dataset sources, curation methodologies, and model architectures.

Table 6: Mean pairwise CWE-type probe AUROC (off-diagonal) across three datasets and three models. PRECISEBUGS and DELTASECOMMITS show strong, consistent CWE separability (86–92%) across all models, confirming functional encoding is architecture-independent.

Dataset	Qwen-7b	CodeLlama-7b	StarCoder2-7b
DELTASECOMMITS	91.5%	91.9%	91.7%
PRECISEBUGS	86.9%	90.1%	89.9%

Pairwise CWE-type probing on PRECISEBUGS replicates across all three 7B models, demonstrating that functional CWE understanding is consistent even on real-world, community-sourced vulnerabilities. The eight CWE types in PRECISEBUGS (CWE-119, CWE-121, CWE-122, CWE-190, CWE-369, CWE-416, CWE-457, CWE-476) span buffer overflows, type confusion, integer overflows, and use-after-free vulnerabilities.

Cross-family vulnerability direction transfer across models. Cross-family vulnerability direction transfer (computed using all five bug families: memory safety, injection, resource management, information disclosure, and control flow) shows consistent universality across models. The vulnerability direction learned from one family generalizes to other families at high rates across all three models, demonstrating that the vulnerability signal encodes properties shared across family boundaries.

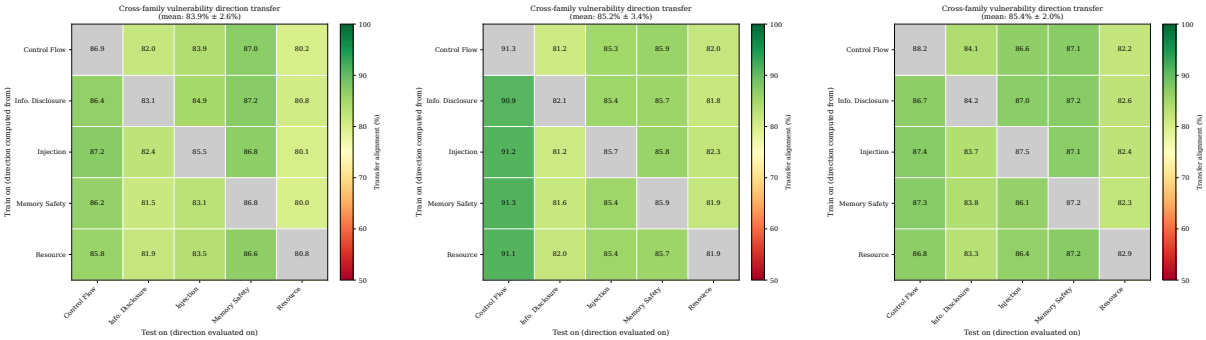


(a) Qwen-7B (mean-token pooling; 85–100% AUROC).

(b) CodeLlama-7B (mean-token pooling; 84–100% AUROC).

(c) StarCoder2-7B (mean-token pooling; 85–100% AUROC).

Figure 9: Pairwise CWE-type probe AUROC on PRECISEBUGS (Layer 19) across three model families. All models exhibit strong separation between CWE types, indicating consistent functional vulnerability encoding across architectures.



(a) Qwen-7B (83.9% $\pm$ 2.6%).

(b) CodeLlama-7B (85.2% $\pm$ 3.4%).

(c) StarCoder2-7B (85.4% $\pm$ 2.0%).

Figure 10: Cross-family vulnerability direction transfer across five bug families (memory safety, injection, resource management, information disclosure, control flow). Diagonal cells (gray) indicate self-transfer. All models show strong transfer, with StarCoder2 achieving the highest mean and lowest variance.

All three models show that vulnerability directions transfer at high rates (83–85% mean) across family boundaries, with consistent patterns: strong transfer to memory safety and information disclosure families (85–87%), weaker transfer to resource management and injection families (80–83%). This pattern is stable across all three model architectures, confirming the universality of the cross-family transfer property.

F Statistical Significance

We report three complementary tests, each covering all three datasets $\times$ 8 models $\times$ seven layers (168 tests each). Test 1 (one-sample) measures whether per-pair projections onto the vulnerability direction are consistently non-zero. Test 2 (paired) measures whether paired (vulnerable, secure) samples occupy a tighter neighbourhood than random unpaired samples. For both, we report the effect size (Cohen's d) in Tables 7 and 8. Statistical significance is established by Test 3, a one-sided binomial test on direction alignment (Table 9): for every model and dataset, the best-performing layer's alignment is far above chance. The three tests establish that a consistent vulnerability direction exists, that paired samples form tighter neighbourhoods than random pairs, and that this alignment is systematic across models, layers, and datasets.

Table 7: **Effect size of the one-sample test on per-pair projections onto the vulnerability direction, per model and layer.** A positive d indicates that vulnerable code projects further along the direction than its secure counterpart. Significance is established by the binomial test (Table 9).

Model	L3	L7	L11	L15	L19	L23	L27
DELTASECOMMITS							
CodeLlama-7B-Instruct	0.305	0.306	0.310	0.316	0.333	0.338	0.346
CodeLlama-13B-Instruct	0.285	0.289	0.298	0.316	0.336	0.366	0.369
DeepSeek-Coder-6.7B-Instruct	0.313	0.321	0.338	0.257	0.356	0.463	0.548
Qwen2.5-7B-Instruct	0.657	0.314	0.317	0.316	0.322	0.329	0.438
Qwen2.5-Coder-7B-Instruct	0.683	0.306	0.323	0.334	0.366	0.406	0.523
Qwen2.5-14B-Instruct	0.556	0.317	0.315	0.318	0.321	0.323	0.320
StarCoder2-7B	0.289	0.295	0.292	0.295	0.303	0.320	0.325
StarCoder2-15B	0.284	0.291	0.297	0.265	0.264	0.274	0.281
PRECISEBUGS							
CodeLlama-7B-Instruct	0.186	0.186	0.187	0.188	0.190	0.191	0.194
CodeLlama-13B-Instruct	0.190	0.190	0.192	0.194	0.196	0.202	0.202
DeepSeek-Coder-6.7B-Instruct	0.199	0.199	0.199	0.142	0.149	0.163	0.190
Qwen2.5-7B-Instruct	0.260	0.187	0.188	0.187	0.188	0.190	0.227
Qwen2.5-Coder-7B-Instruct	0.260	0.190	0.195	0.195	0.200	0.214	0.246
Qwen2.5-14B-Instruct	0.235	0.187	0.187	0.187	0.187	0.187	0.187
StarCoder2-7B	0.192	0.194	0.193	0.194	0.196	0.201	0.204
StarCoder2-15B	0.192	0.192	0.193	0.188	0.184	0.188	0.193
SVEN							
CodeLlama-7B-Instruct	0.178	0.178	0.180	0.182	0.186	0.185	0.188
CodeLlama-13B-Instruct	0.174	0.176	0.180	0.187	0.197	0.201	0.200
DeepSeek-Coder-6.7B-Instruct	0.169	0.170	0.173	0.170	0.181	0.191	0.207
Qwen2.5-7B-Instruct	0.308	0.160	0.161	0.161	0.164	0.164	0.211
Qwen2.5-Coder-7B-Instruct	0.322	0.159	0.168	0.174	0.184	0.199	0.318
Qwen2.5-14B-Instruct	0.271	0.160	0.160	0.161	0.162	0.163	0.163
StarCoder2-7B	0.182	0.187	0.188	0.189	0.193	0.200	0.202
StarCoder2-15B	0.178	0.181	0.183	0.172	0.166	0.175	0.178

Table 8: **Effect size of the paired test comparing within-pair against cross-pair cosine similarity, per model and layer.** Larger d indicates that paired (vulnerable, secure) samples occupy a tighter neighbourhood than random unpaired samples. Significance is established by the binomial test (Table 9).

Model	L3	L7	L11	L15	L19	L23	L27
DELTASECOMMITS							
CodeLlama-7B-Instruct	0.524	0.802	1.162	1.620	2.824	3.654	3.430
CodeLlama-13B-Instruct	0.676	0.976	1.529	1.927	3.018	4.553	3.921
DeepSeek-Coder-6.7B-Instruct	0.947	1.400	1.957	2.529	3.426	3.704	3.828
Qwen2.5-7B-Instruct	2.696	0.676	0.877	0.943	0.920	1.606	3.768
Qwen2.5-Coder-7B-Instruct	2.753	1.898	2.792	2.535	2.259	3.422	3.567
Qwen2.5-14B-Instruct	2.474	0.653	0.846	0.878	0.903	0.919	0.969
StarCoder2-7B	1.238	1.678	2.197	2.142	2.236	3.035	3.350
StarCoder2-15B	1.089	1.479	2.439	2.428	2.564	2.841	3.576
PRECISEBUGS							
CodeLlama-7B-Instruct	0.722	0.836	1.047	1.198	1.666	2.252	2.266
CodeLlama-13B-Instruct	0.754	0.881	1.200	1.348	1.780	2.355	2.546
DeepSeek-Coder-6.7B-Instruct	0.779	0.952	1.129	1.417	1.842	2.368	2.428
Qwen2.5-7B-Instruct	1.654	0.878	1.084	1.130	1.071	1.433	2.347
Qwen2.5-Coder-7B-Instruct	1.667	1.320	1.931	1.810	1.628	2.405	2.219
Qwen2.5-14B-Instruct	1.562	0.804	0.967	0.977	1.028	1.051	1.097
StarCoder2-7B	1.129	1.420	1.689	1.477	1.368	1.872	1.985
StarCoder2-15B	0.968	1.157	1.580	1.627	1.640	1.759	2.306
SVEN							
CodeLlama-7B-Instruct	0.972	1.252	1.581	1.871	3.068	3.575	3.056
CodeLlama-13B-Instruct	0.958	1.145	1.844	2.437	3.247	4.215	4.304
DeepSeek-Coder-6.7B-Instruct	0.957	1.241	1.585	2.272	3.081	3.572	3.412
Qwen2.5-7B-Instruct	2.221	1.245	1.599	1.747	1.659	2.153	3.140
Qwen2.5-Coder-7B-Instruct	2.279	1.813	2.925	3.029	2.757	3.617	2.922
Qwen2.5-14B-Instruct	2.302	1.051	1.385	1.336	1.428	1.599	1.461
StarCoder2-7B	1.585	2.437	3.170	2.332	2.099	2.728	2.961
StarCoder2-15B	1.387	1.757	2.797	2.933	2.857	2.960	3.433

Table 9: **One-sided binomial test on direction alignment.** Smallest (best-case) p-value across the 7 probed layers, per model and dataset. For every model and dataset, the best-performing layer's alignment is far above chance.

Model	DELTASECOMMITS	PRECISEBUGS	SVEN
CodeLlama-7B-Instruct	2.6×10^{-294}	6.1×10^{-200}	7.2×10^{-25}
CodeLlama-13B-Instruct	$< 10^{-300}$	5.0×10^{-215}	1.7×10^{-23}
DeepSeek-Coder-6.7B-Instruct	$< 10^{-300}$	9.0×10^{-195}	7.2×10^{-25}
Qwen2.5-7B-Instruct	$< 10^{-300}$	4.3×10^{-248}	1.7×10^{-32}
Qwen2.5-Coder-7B-Instruct	$< 10^{-300}$	4.7×10^{-258}	8.3×10^{-30}
Qwen2.5-14B-Instruct	$< 10^{-300}$	2.0×10^{-273}	2.1×10^{-31}
StarCoder2-7B	$< 10^{-300}$	1.2×10^{-184}	6.1×10^{-32}
StarCoder2-15B	$< 10^{-300}$	2.6×10^{-204}	2.1×10^{-31}

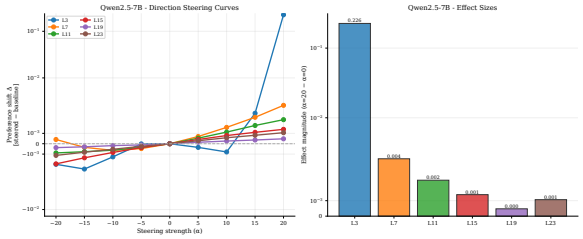
G Steering Causal Validation

Causal validation: does manipulating the direction change model behavior? This appendix attempts to showcase some plausible effect of steering the vulnerability direction throughout layers. We do not, however, claim this protocol to be the correct steering method, specially since there are cascading effects if applied at early layers. Some LLMs do not exhibit a clean chance under this protocol. Deeper analysis using other steering methods remains to be explored.

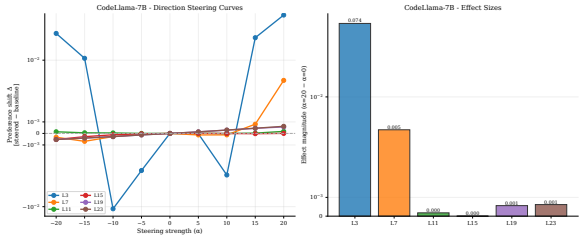
Steering protocol. We applied directional steering to 100 real vulnerable–secure code pairs from DELTASECOMMITS. Each pair is presented in C-code context with three prompting prefixes ("// C code:", "// Function:", "// Implementation:") to reduce prompt artifacts. Preference is measured as the difference in per-token log-probabilities: $\log P_{\text{secure}} - \log P_{\text{vulnerable}}$. Steering is applied as an activation-space perturbation at layer output: $h_i' = h_i - \alpha \cdot \hat{d}$, where $\alpha \in [-20, -10, 0, 10, 20]$ and $\hat{d}$ is the normalized vulnerability direction.

Causal observations: We replicate directional steering across Qwen2.5-7B-Instruct and CodeLlama-7B-Instruct on the full C-code DELTASECOMMITS subset. We focus our analysis on **layers L7–L23** to ensure representational stability; **Layer 3** exhibits higher variance due to its position early in the forward pass. Across **layers L7–L23**, suppressing the vulnerability direction produces a consistent effect on model preference for secure code (Figure 11).

Causal interpretation: Intervening on the hidden representation of the model, deviates its preference to higher probability of secure or insecure variants, according to the direction of the perturbation.



(a) Qwen2.5-7B-Instruct



(b) CodeLlama-7B-Instruct

Figure 11: Direction steering in deeper layers (L7–L23): negligible causal effects despite geometric coherence. Preference is measured as $\log P(\text{secure}) - \log P(\text{vulnerable})$. Across layers L7–L23, suppressing the vulnerability direction (increasing α from -20 to $+20$) produces small preference shifts in both models.

Finding. Direction steering confirms that the vulnerability direction meaningfully influences the model's relative safety judgments. We note, however, that this experiment is only applicable to a subset of models. Different works in literature have explored steering and interventions in a variety of ways, including, but not limited to, KV-cache steering (Belitsky et al., 2025), or direct weight updates (Adila et al., 2026). The specific method of steering should be tested to each model independently. This experiment validates the vulnerability direction in Qwen2.5-7B-Instruct and CodeLlama-7B-Instruct, but further work is needed to confirm which intervention methods are effective and consistent in other models.

H Comprehensive Model Analysis: All Heatmaps and Figures

This appendix presents complete heatmap results for all 8 models across all vulnerability metrics, datasets, and analysis types.

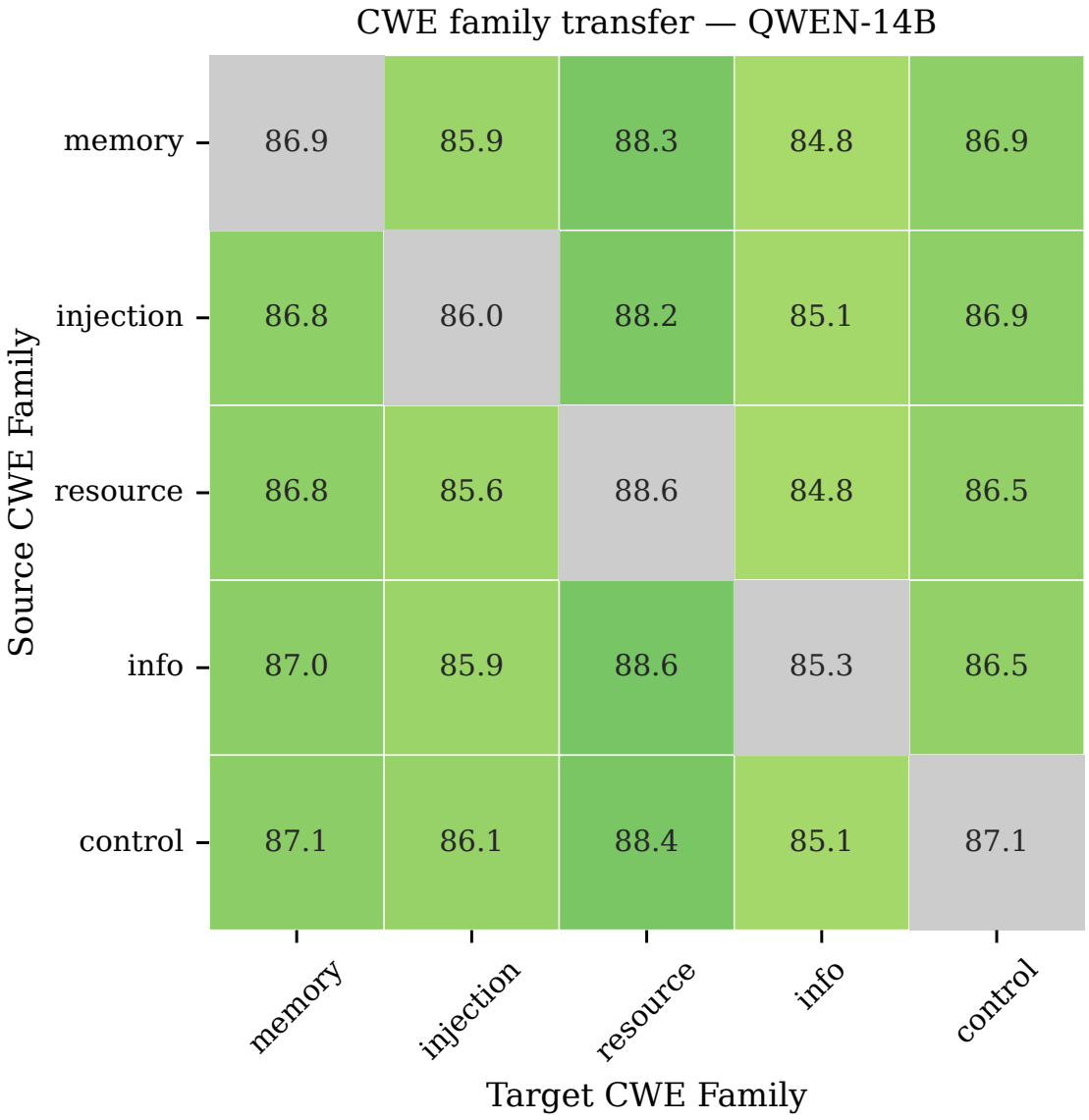


Figure 12: CWE-pairwise alignment: Qwen2.5-14B-Instruct (scaled model). The pattern is consistent across the tested scales (6.7B–15B).

H.1 Multi-Model Comparisons

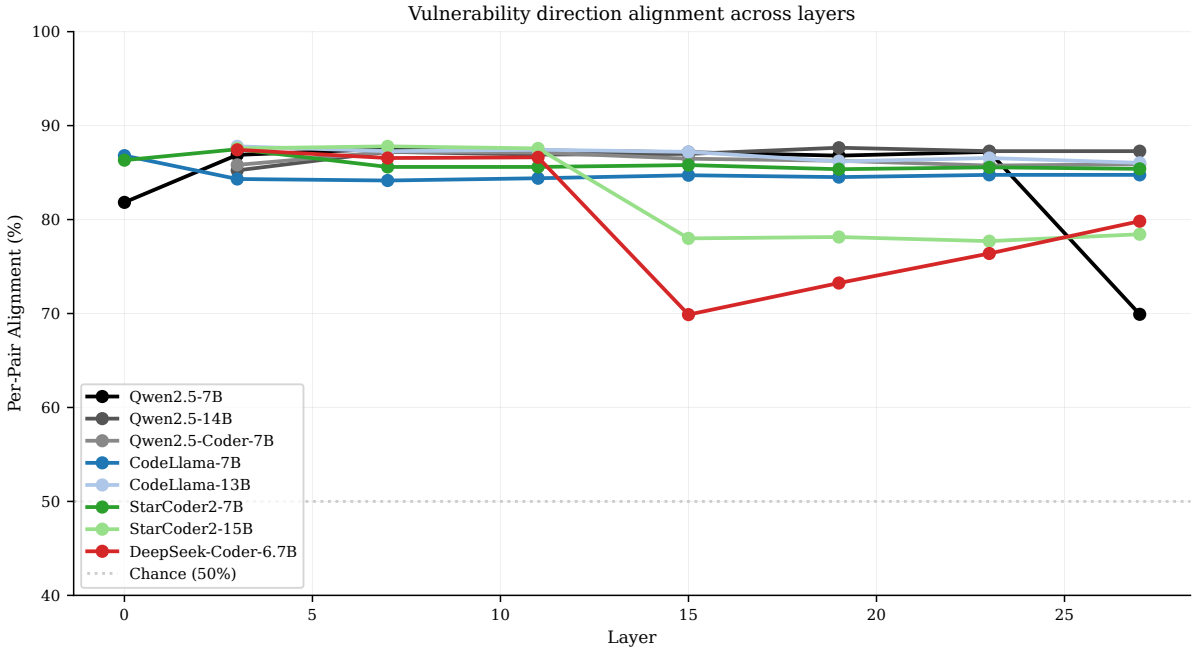


Figure 13: Direction alignment combined: All models overlaid. Shows convergence across architectures.

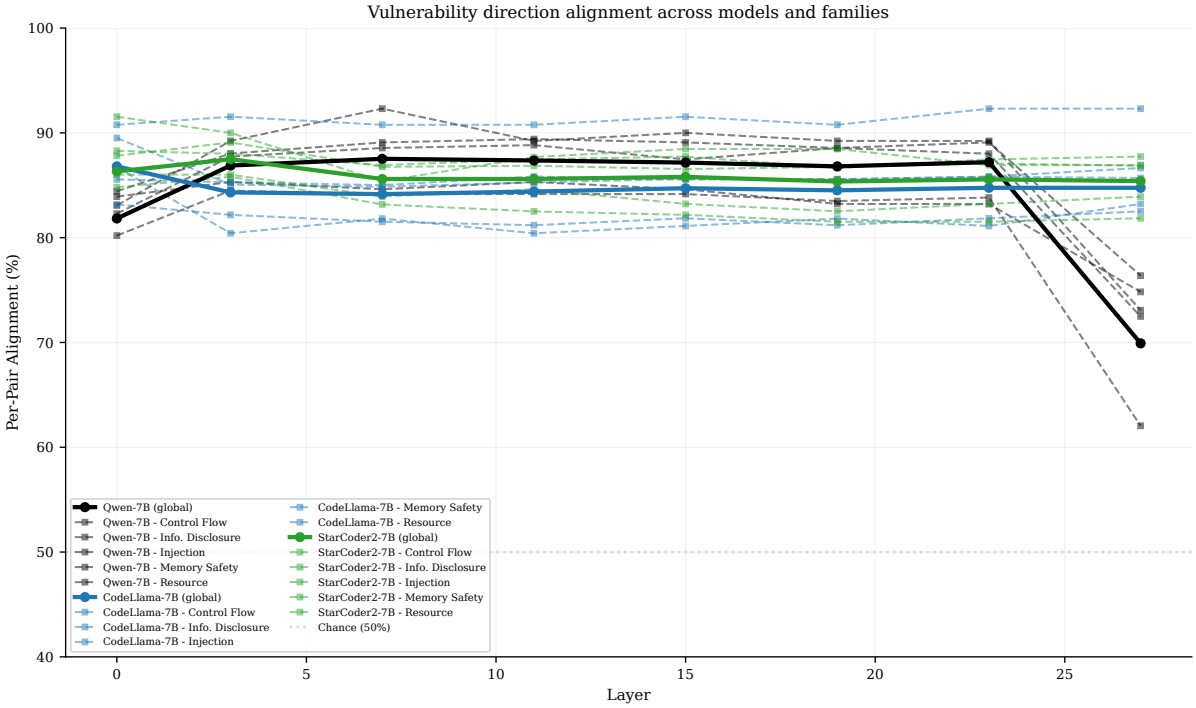


Figure 14: Direction alignment by family: Three architectural families show nearly identical patterns.

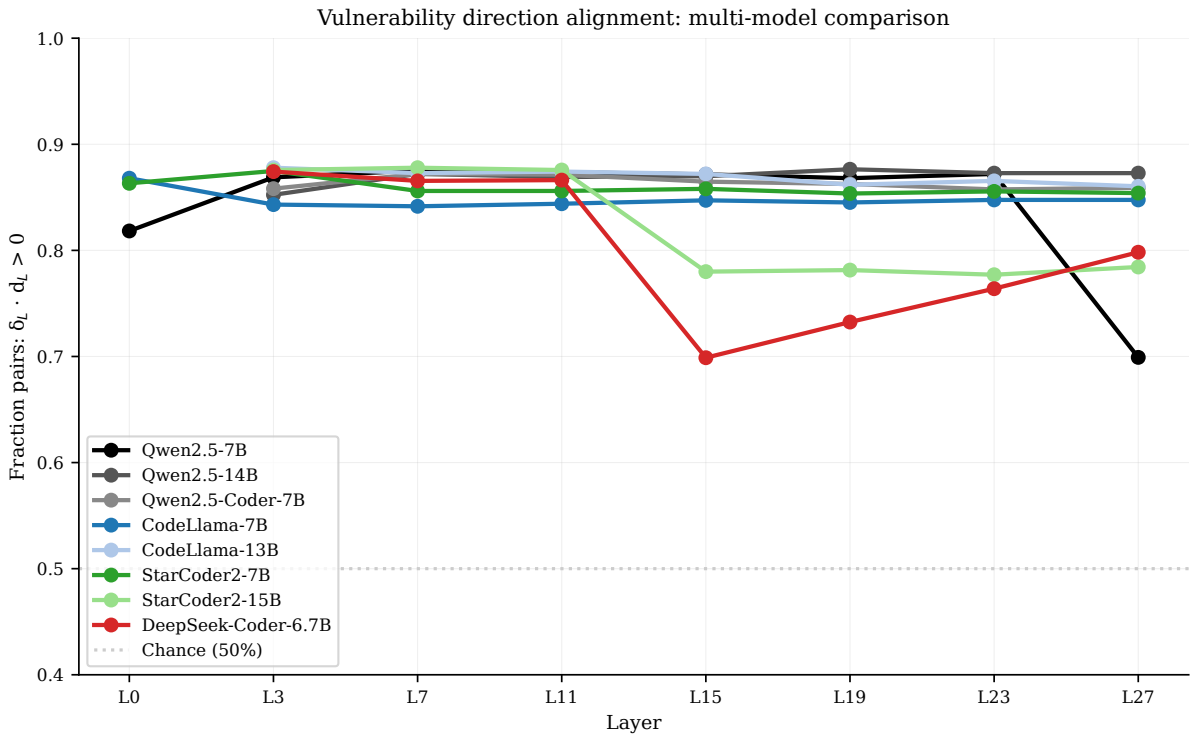


Figure 15: Multi-model alignment comparison: all 8 models side-by-side, showing consistent patterns across the tested scales (6.7B–15B).

H.2 Cross-Family Transfer Heatmaps

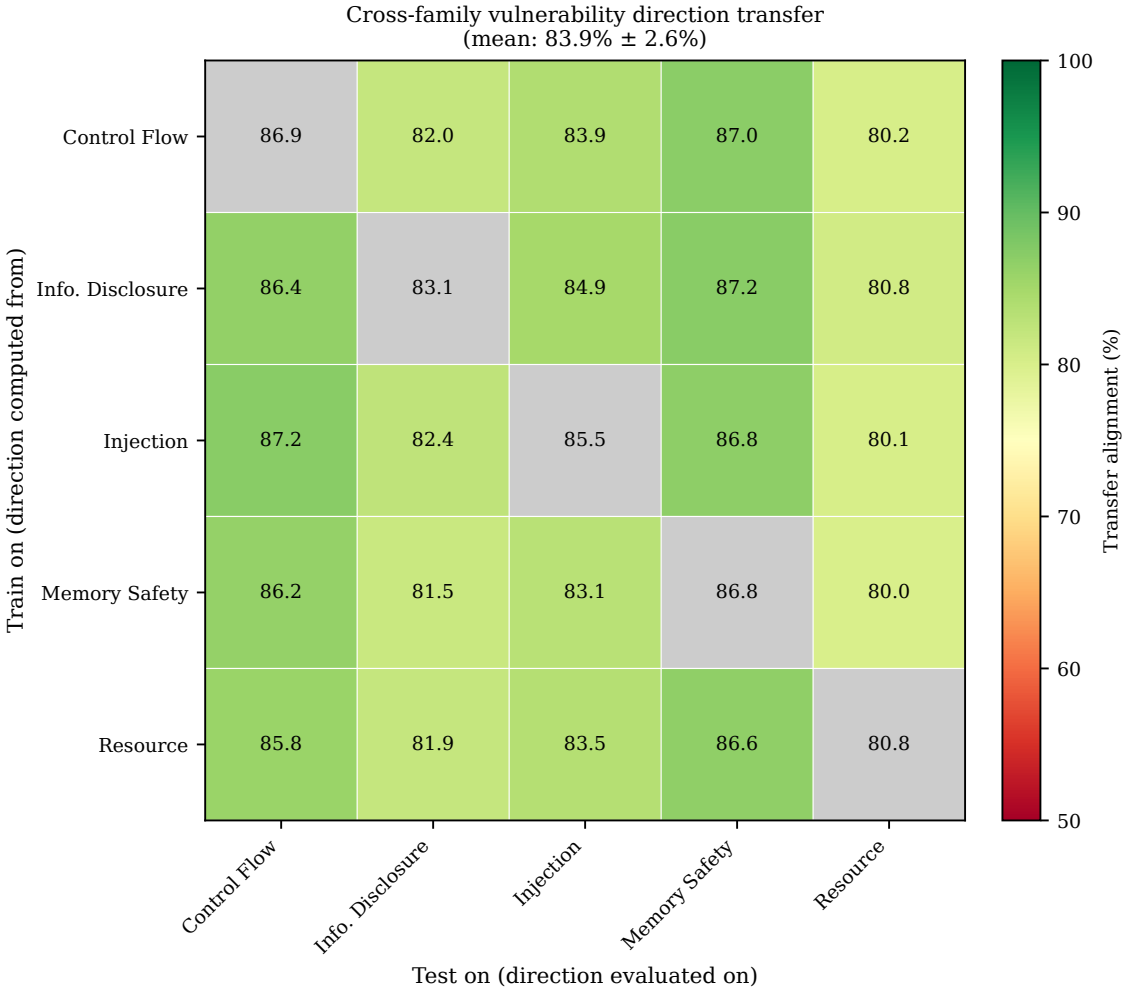


Figure 16: Cross-family transfer: Qwen2.5-7B-Instruct. Warm colors (80-84%) show strong transfer; cooler colors (70-75%) show weaker injection transfer.

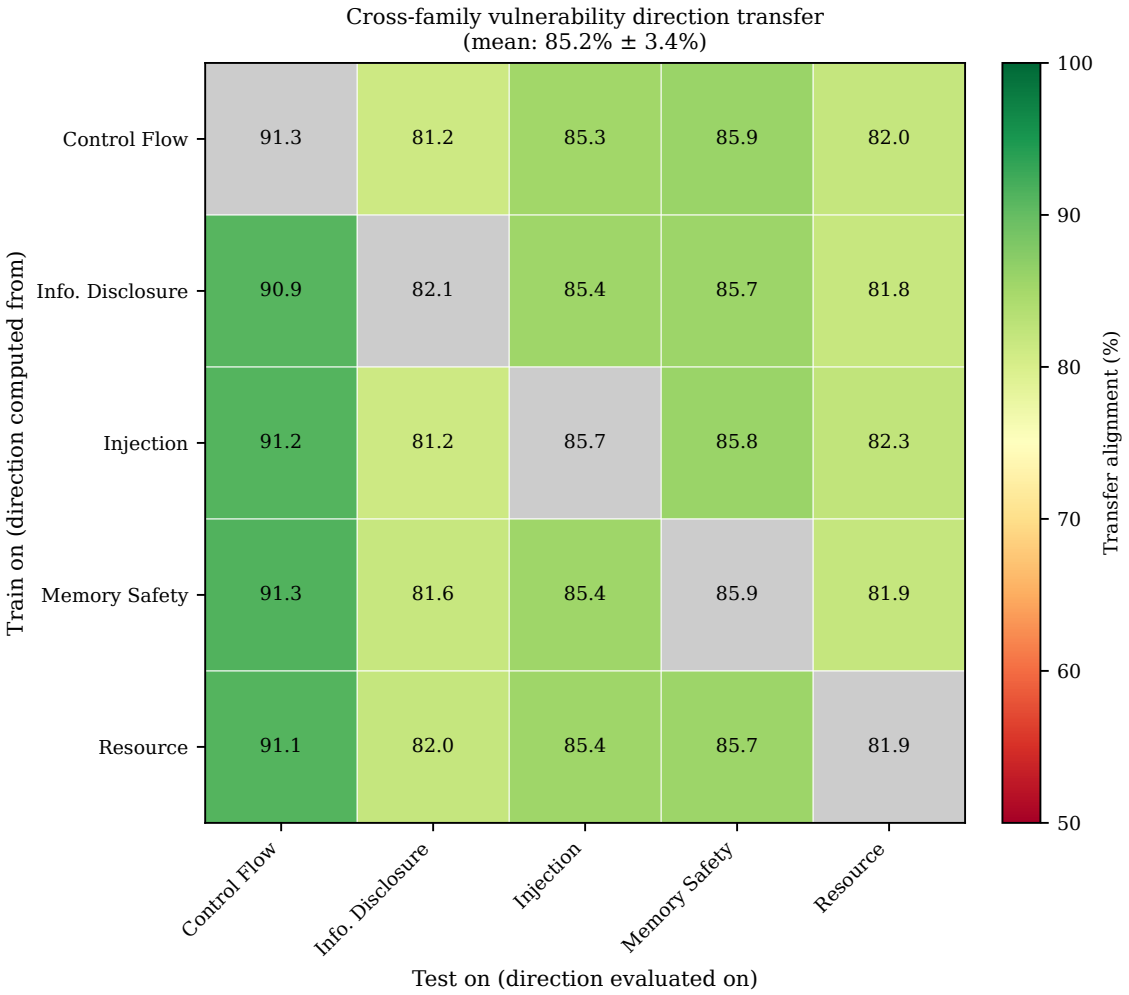


Figure 17: Cross-family transfer: CodeLlama-7B-Instruct.

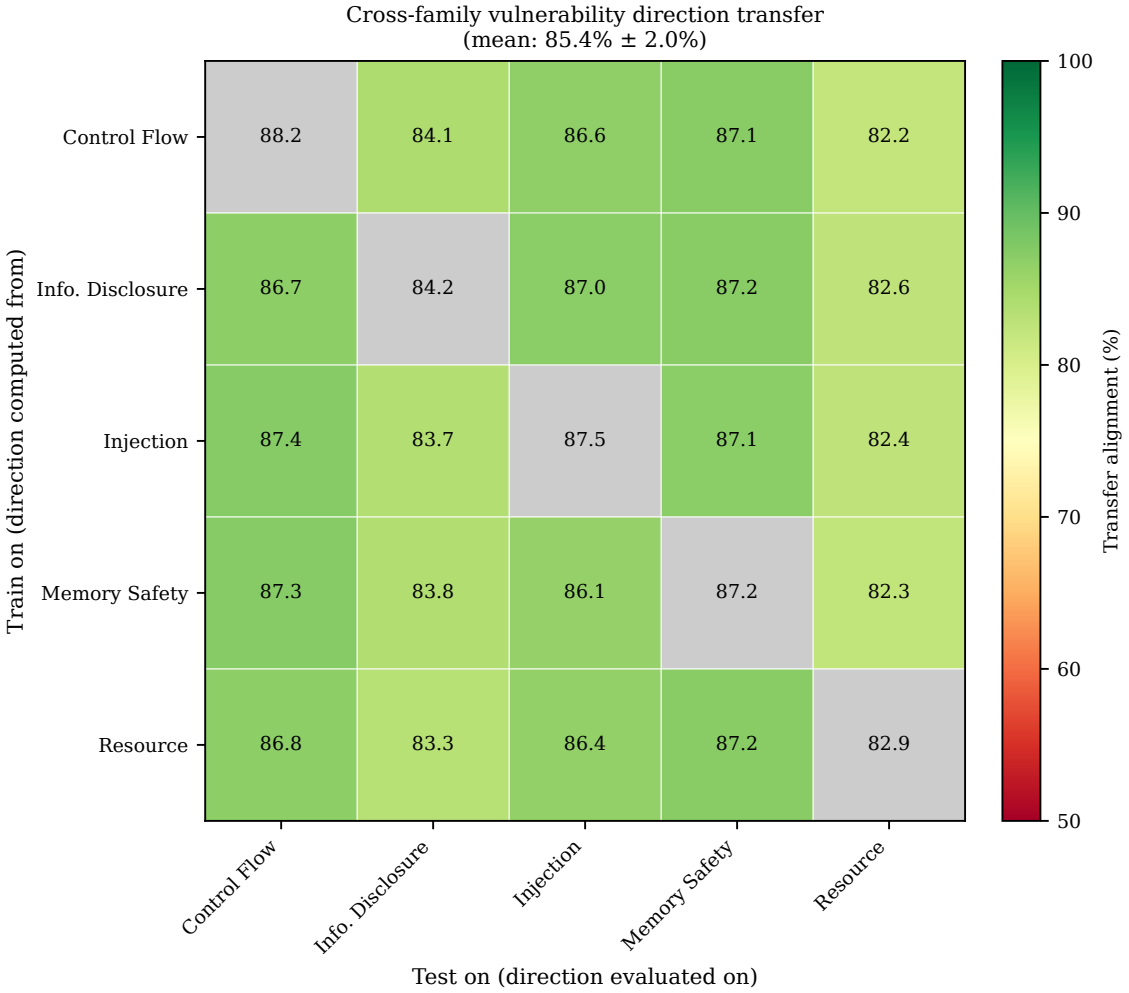


Figure 18: Cross-family transfer: StarCoder2-7B.

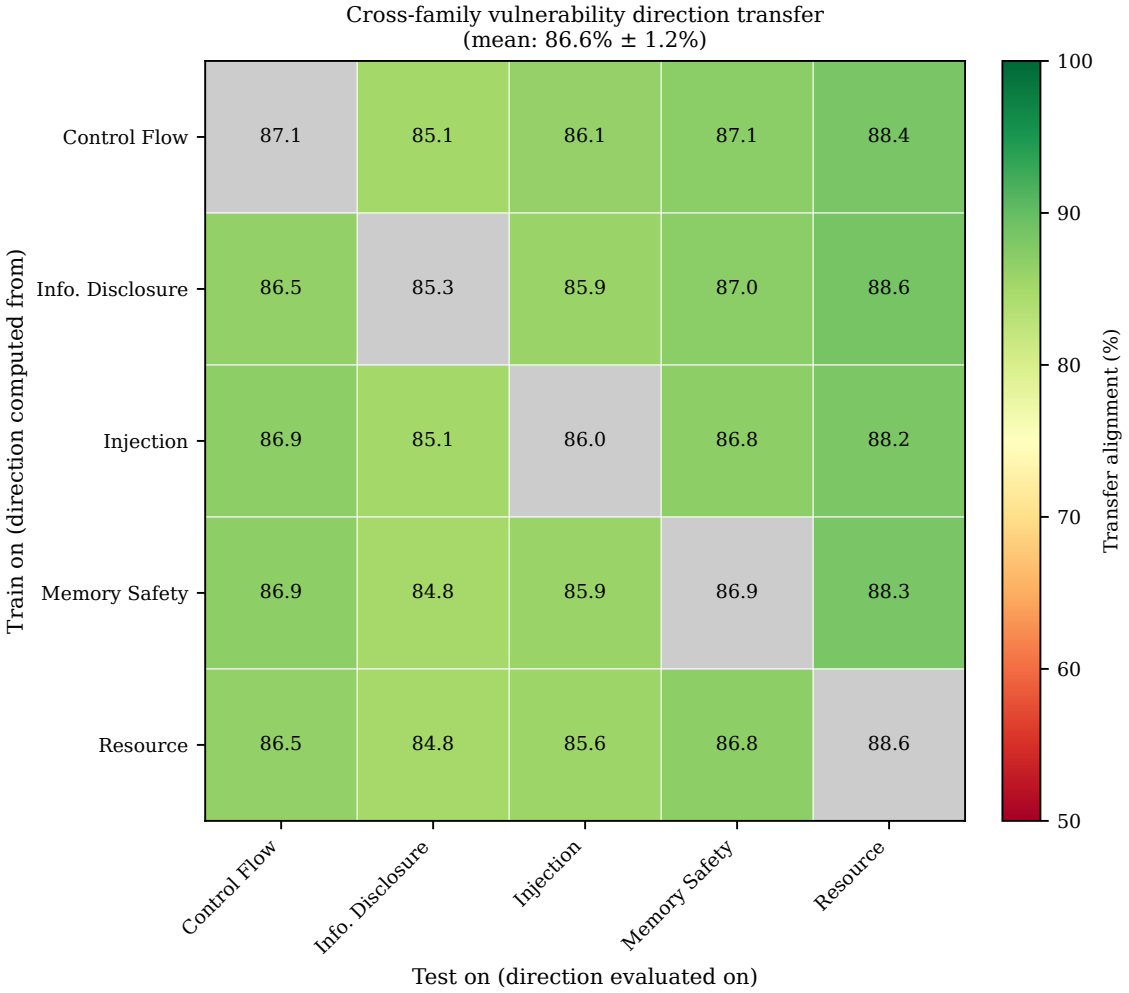


Figure 19: Cross-family transfer: Qwen2.5-14B-Instruct. Transfer patterns consistent across scale.

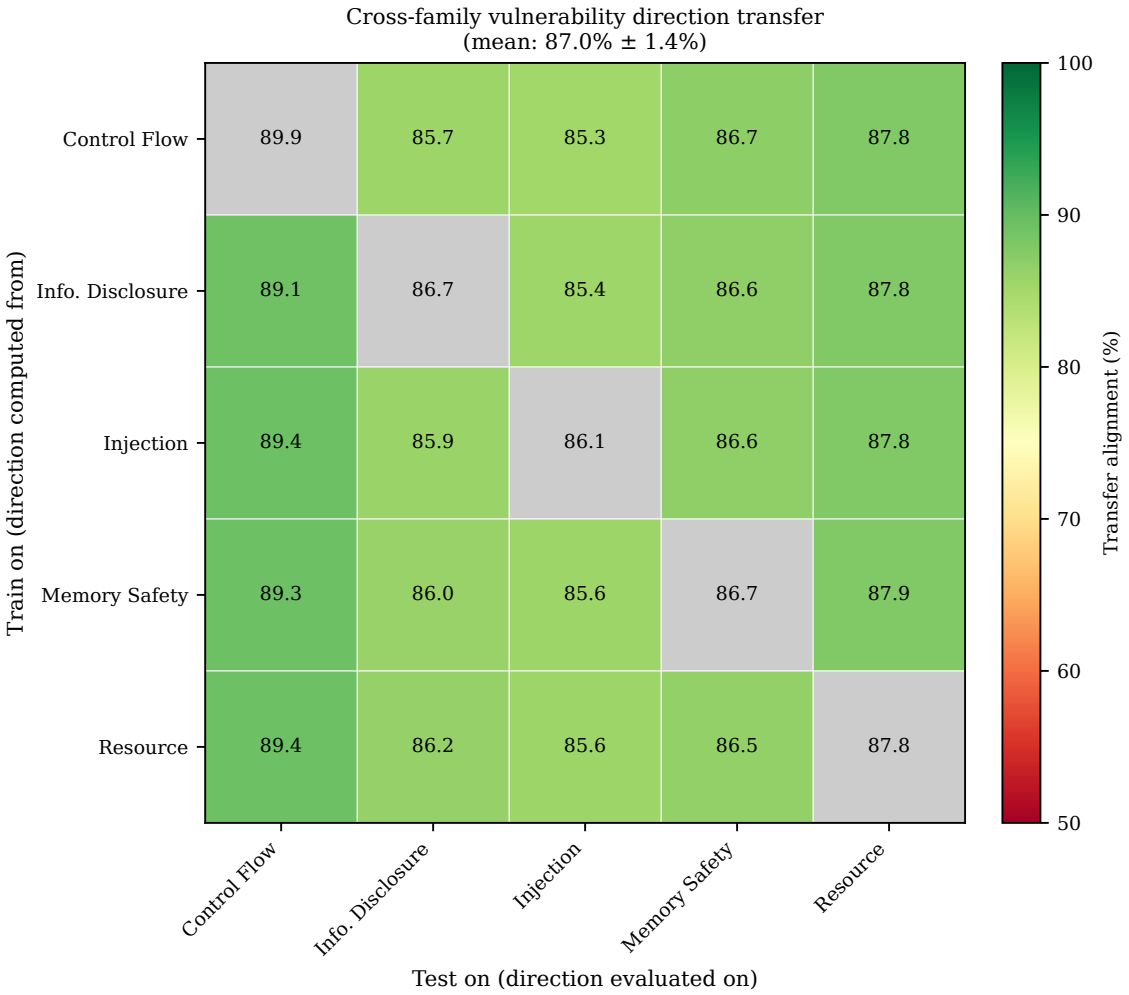


Figure 20: Cross-family transfer: CodeLlama-13B-Instruct.

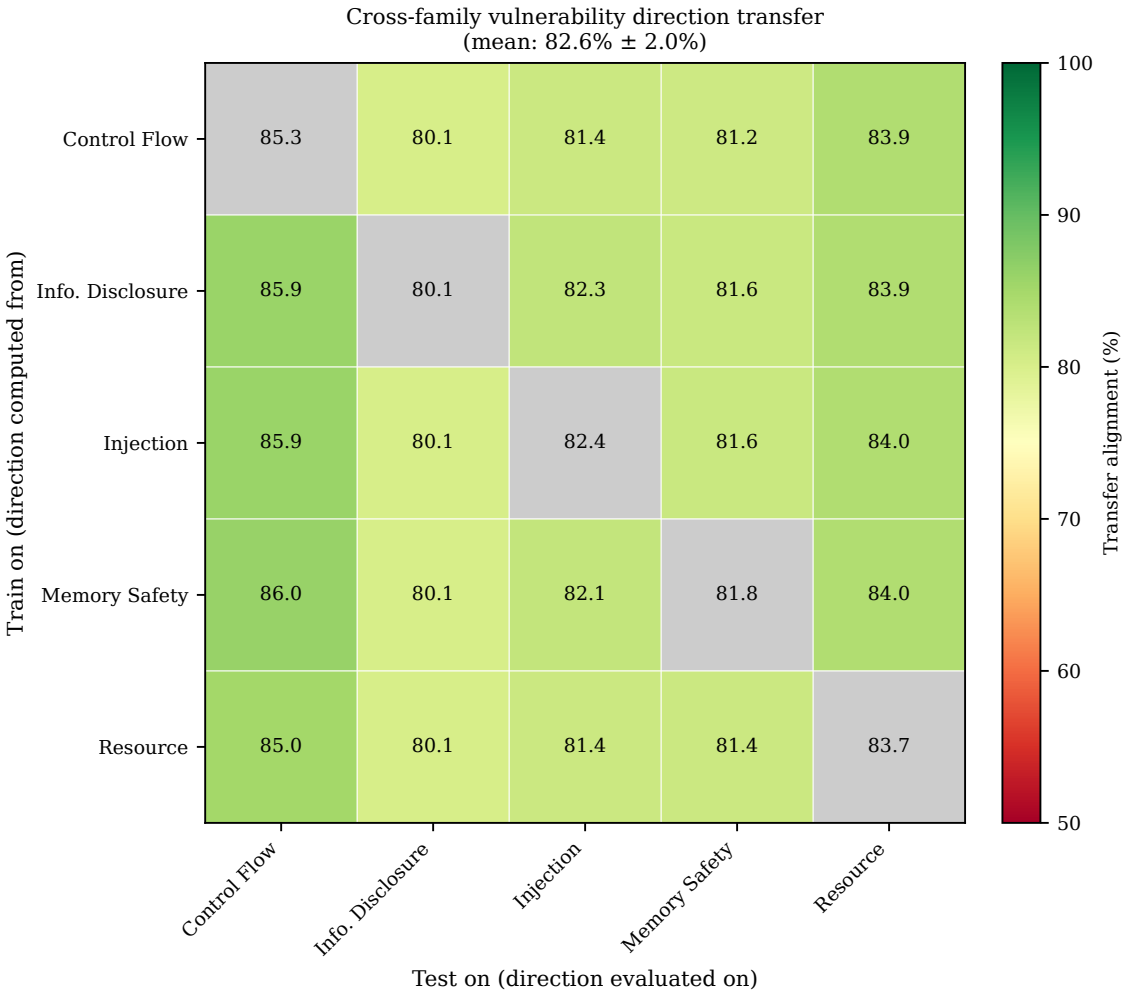


Figure 21: Cross-family transfer: StarCoder2-15B.

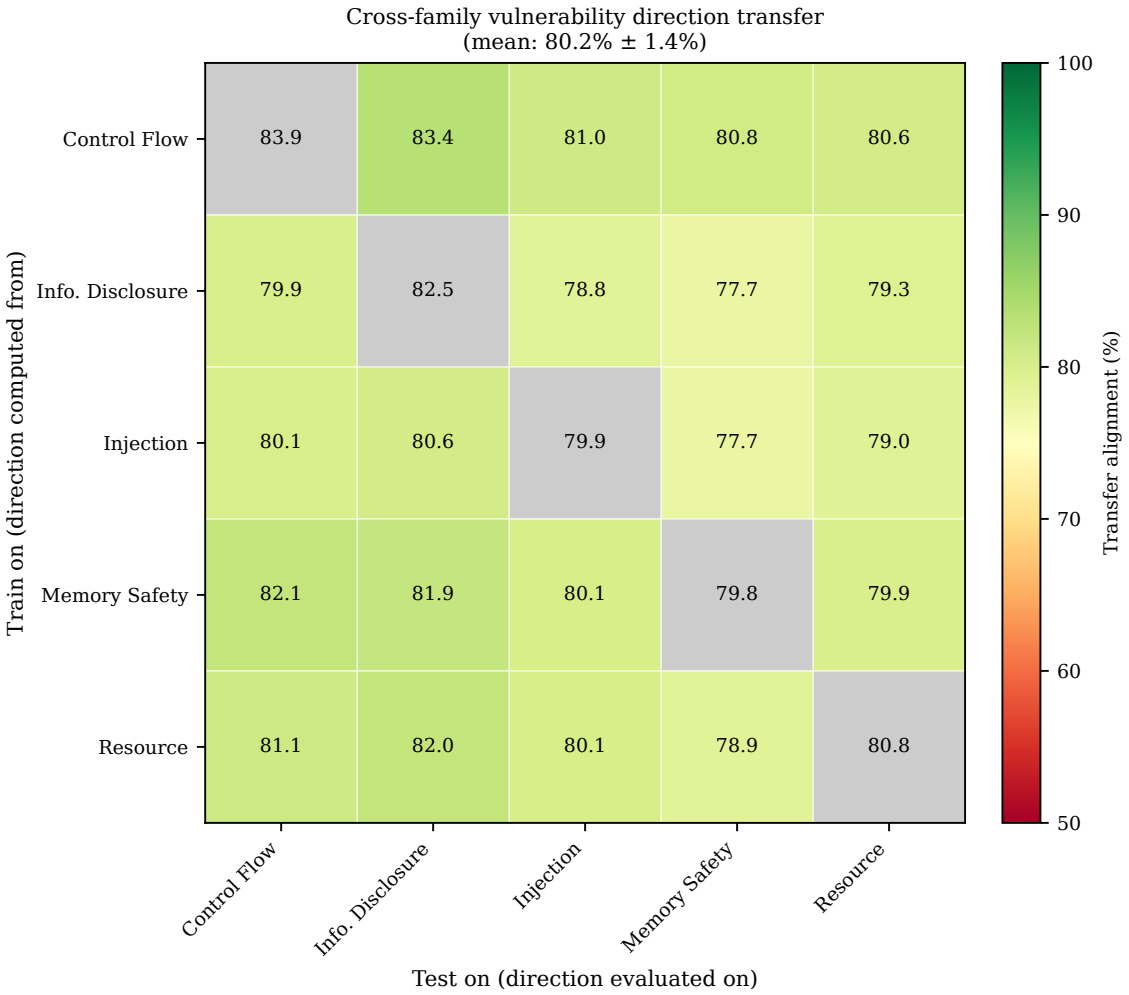


Figure 22: Cross-family transfer: DeepSeek-Coder-6.7B-Instruct.

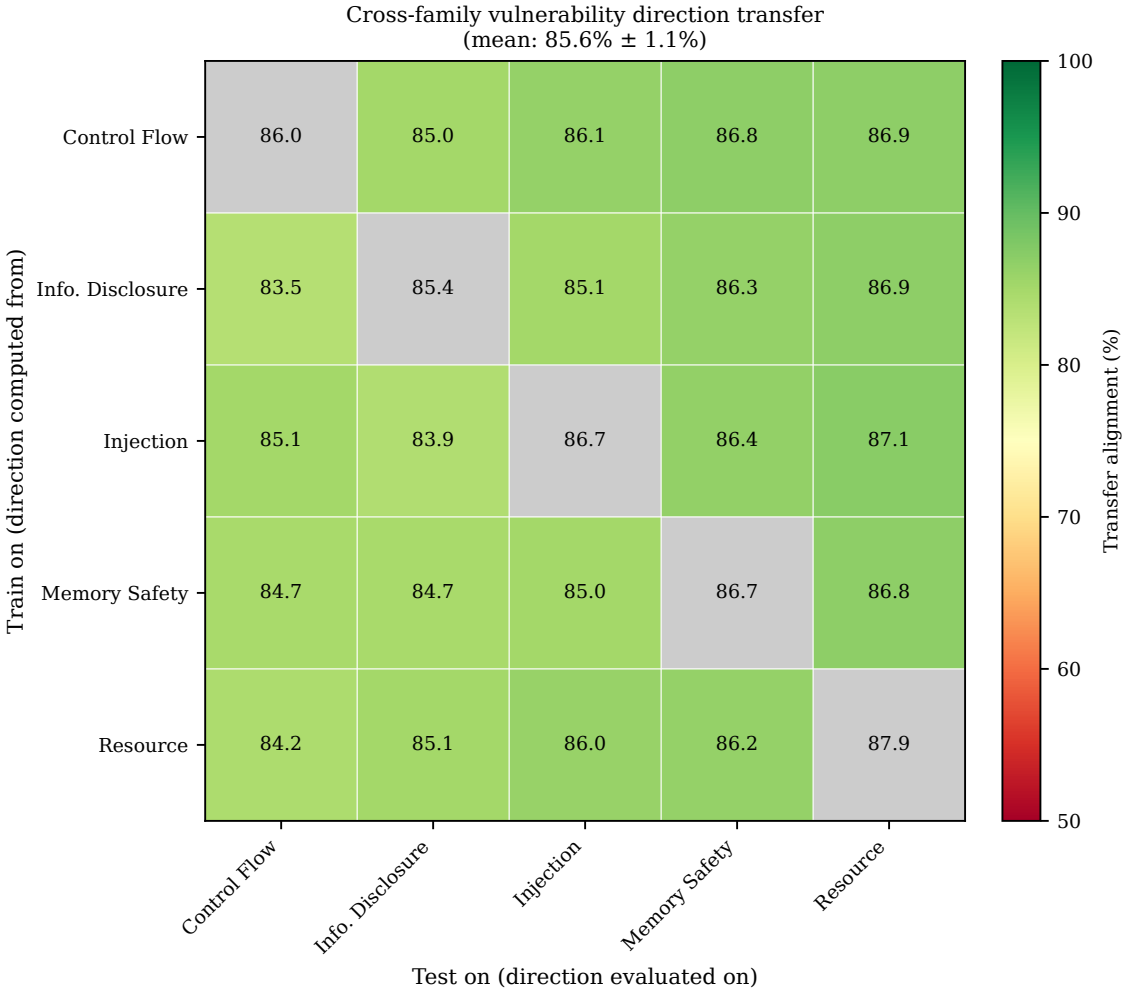


Figure 23: Cross-family transfer: Qwen2.5-Coder-7B-Instruct.