Do Language Models Prefer Vulnerable Code? A Probabilistic Study of Insecure Code Preference

Rui Melo[1,2], Sofia Reis[2], Andre Catarino[2], Rui Abreu[2,3]
[1]*Carnegie Mellon University*, Pittsburgh, PA, USA
[2]*Faculty of Engineering, University of Porto*, Porto, Portugal
[3]*INESC-ID*, Porto, Portugal

***Abstract*—Large Language Models (LLMs) are increasingly integrated into software development and testing workflows, offering the promise of automated code generation, test synthesis, and program repair. However, ensuring the security of LLM-generated code remains a critical challenge for software verification and validation, as these models may inadvertently learn and propagate insecure patterns from their training data. In this paper, we present a probabilistic testing framework for evaluating the security alignment of code LLMs, analyzing their internal behavior across three dimensions: *fluency* (does the code appear natural?), *preference* (which version is the model more likely to generate?), and *confidence* (how certain is the model about its choice?). Using `Delta-Secommits`, a 2,422 real-world vulnerability-patch pairs spanning 25 CWE categories, we conduct the first empirical study of how code LLMs probabilistically favor secure versus insecure code. Our results reveal a significant security misalignment: LLMs exhibit a bias toward insecure code in approximately 92% of cases. Even when secure code is as fluent or confidently predicted, models still prefer the vulnerable version in the vast majority of comparisons.**

For researchers, our findings extend existing evaluation frameworks by introducing probabilistic security alignment, measuring not only generated outputs, but also the likelihoods that drive them. For tool builders, the implication is clear: AI coding assistants must be designed for and tested against secure defaults, or they risk amplifying vulnerabilities at scale.

https://huggingface.co/datasets/rufimelo/DeltaSecommits

https://github.com/rufimelo99/lm-insecure-bias

https://softwareheritage.org

***Index Terms*—Large Language Models, Security Code Vulnerabilities, AI-assisted Development**

I. Introduction

The rise of Large Language Models (LLMs) is transforming software development, not only through automated code generation, but also by reshaping software testing, verification, and quality assurance pipelines [1], [2]. However, as LLMs become integral to development and testing workflows, their tendency to propagate insecure patterns learned from training data raises critical concerns for software validation and security testing [3]–[9]. Despite their productivity benefits [1], LLMs must be rigorously validated before deployment. Assuming that LLMs inherently generate secure code is dangerous: insecure suggestions from user-facing assistants (e.g., GitHub Copilot, ChatGPT) can silently propagate into production systems, leading to exploits, compliance violations, and system failures. This raises the risk that productivity gains come at the cost of security misalignment. For the software testing and verification community, this risk is especially acute, as insecure LLM-generated code can be integrated at scale, undermining trust in AI-assisted development.

Reactive methods, such as static and dynamic analysis [10] and manual review [3], identify vulnerabilities after code is generated, acting as post-hoc verification steps. *Proactive techniques*, such as prompt engineering [11]–[14], attempt to steer models toward secure outputs but often fail to address underlying probabilistic biases. In the context of software testing, relying solely on these methods creates a cycle of detection and repair that increases testing overhead and delays release cycles.

At their core, LLMs generate code via token-level probability assignments. From a testing perspective, these probabilities define the space of likely program behaviors that verification and validation tools must contend with. If insecure variants are assigned higher likelihood, unsafe code becomes the model's default—posing a critical challenge for automated testing and verification systems that assume model outputs are trustworthy. Prior work shows that LLMs can favor insecure patterns even when secure alternatives exist [1], [8], [9]. This misalignment can directly impact test generation, code review automation, and vulnerability detection.

Unlike prior work that focuses on output correctness or vulnerability detection [8]–[10], [15]–[25], our work introduces a probabilistic testing lens that exposes internal model biases invisible to traditional testing benchmarks. By diagnosing these biases, we provide a foundation for understanding how LLMs probabilistically favor insecure code. This analysis can guide the development of testing frameworks and verification tools to better assess the security risks of LLM-generated code.

Thus, to guide our research, we ask: **(RQ1) How can we operationalise security alignment in LLMs through probabilistic metrics such as fluency, preference, and uncertainty?** and **(RQ2) Do state-of-the-art base LLMs exhibit security misalignment when comparing secure and vulnerable code?**

To investigate this issue, we propose a probabilistic testing framework based on three behavioral dimensions: **fluency**, how *natural* or familiar a code variant appears to the model; [1] **preference**, which variant it is more likely to produce; and

[1]We use the term "behavioural" since they capture observable outputs, not internal mechanisms, reflecting responses to secure and insecure alternatives.

confidence, how certain it is about that choice. While these concepts are well-studied in Natural Language Processing (NLP) [26]–[31], their joint use as test oracles for code security has not been explored. We hypothesise that security-aligned LLMs should assign higher fluency, preference, and confidence to secure code than to insecure alternatives.

We empirically evaluate this hypothesis across multiple Common Weakness Enumeration (CWE) categories using state-of-the-art open-source code LLMs, including `CodeLlama` (7B, 13B), `StarCoder2` (3B, 7B), `DeepSeek-Coder` (6.7B), and `Mellum-4B`. Our analysis is grounded in a dataset of real vulnerability-fixing commits. To support reproducibility and future research, we release this dataset, *`Delta-Secommits`*, comprising 2,422 unique vulnerability-fix pairs mapped to 25 different CWE categories.

Contributions. This paper makes the following contributions:

- **A Probabilistic Testing Framework for Security Alignment:** We introduce a methodology that evaluates LLMs along three dimensions—fluency, preference, and confidence—providing a deeper, model-internal view of security misalignment that complements traditional testing and verification techniques.
- **An Empirical Study Across 25 CWE Categories:** Using a dataset of 2,422 vulnerability–patch pairs, we quantify how often modern LLMs favor insecure code. We find that models are fully aligned with secure code in only 8% of cases, revealing a systemic bias that undermines trustworthy AI-assisted code generation.

The remainder of this paper is structured as follows: Section II presents a motivating example. Section IV describes the dataset, models, and metrics. Section V reports empirical results. Section VI discusses implications for testing and verification. Section III reviews related work, Section VII outlines threats to validity, and Section VIII concludes.

II. Motivation

To illustrate the type of phenomenon our study uncovers, consider the CWE-120 vulnerability shown in Figure 1.

The patched version performs a bounds check before sub-array handling, eliminating the unsafe execution path. Yet `DeepSeek-Coder-6.7B` assigns lower fluency and weaker preference scores (higher perplexity, lower log-probability) to the secure variant, instead favouring the vulnerable code. This misalignment is not an isolated anomaly: as we show later, such cases occur in over 90% of vulnerability-fix pairs in our dataset, even when the secure change is syntactically simple.

This example highlights two points. First, it shows how LLMs can systematically lean toward insecure code, which is problematic for safe code generation. Second, it shows why a pipeline like ours is needed: these training biases are invisible if we only look at final model outputs. By measuring internal probabilistic signals–fluency, preference, and confidence–we can make these hidden tendencies explicit and quantify them across thousands of real-world cases. In this way, the example motivates not just secure code generation, but also the value of our probabilistic framework for studying alignment.

III. Related Work

Bhatt *et al.* [32] introduce *CYBERSECEVAL*, assessing LLMs for insecure code generation and responses aiding cyberattacks. Across 50 CWE types, LLMs produced insecure code in 30% of cases, with advanced models often more vulnerable. Wan *et al.* [33] extend this to *CYBERSECEVAL 3*, evaluating Llama 3 (405B, 70B, 8B), GPT-4 Turbo, and Qwen 2. Llama 3 (405B) generated insecure code in 31% of cases; guardrails like *CodeShield* and *PromptGuard* reduced risks. Hajipour *et al.* [34] present *CodeLMSec*, detecting vulnerabilities in black-box code generation via few-shot prompting across 15 CWE types. Across 2,000 Python and C samples, ChatGPT code was 39% more vulnerable than CodeGen. Wang *et al.* [35] propose *SecuCoGen*, 180 samples covering 21 CWEs. GitHub Copilot generated insecure code 40% of the time, ChatGPT only produced secure code in 5 programs; CWE-aware prompts improved GPT-3.5 security by 175%. *LLMSecCode* [36] provides an open-source framework for code generation and repair. Decoding parameters (e.g., temperature, top-p) can shift security outcomes by 10%, and high correctness does not guarantee secure code, underscoring the need for standardized, community-driven security benchmarks.

A growing body of research has explored benchmarking the security implications of LLMs in code generation. Siddiq *et al.* [37] propose *SALLM*, a framework that systematically benchmarks the security of LLM-generated Python code using a manually curated dataset and a suite of static and dynamic analysis techniques. Their evaluation of five LLMs found that while GPT-4 achieved the highest functional correctness, it was not the most secure; StarCoder exhibited the lowest rate of vulnerabilities, underscoring the necessity of security-focused metrics beyond traditional accuracy.

*a) **Comparison with Prior Work**:* Our findings extend and contextualize prior studies on the security risks of LLM-generated code. Siddiq *et al.* [37] reported that over 50% of model-produced code samples were insecure, while Bhatt *et al.* [32] observed insecure suggestions in roughly 30% of cases. Similarly, Pearce *et al.* [9] found that 40% of Copilot's outputs contained CWE Top 25 vulnerabilities. These results are consistent with a broader body of work highlighting systematic weaknesses in AI-assisted coding [3], [4], [8], [38].

In contrast, our probabilistic, CWE-specific analysis uncovers a deeper issue: models display a strong internal bias toward insecure code, favoring it about **92%** of the time. Strikingly, this preference persists even when the models produce more fluent or confident generations for the secure alternative.

IV. Methodology

Our study systematically investigates how LLMs internally handle secure versus insecure code by analysing three behavioural dimensions—*fluency*, *preference*, and *confidence*—which we operationalise using probabilistic metrics: **Perplexity (PPL)** for *fluency*, **Log Probability (LogProb)** for *preference*, and **Uncertainty** for *confidence*.

While each of these metrics captures a distinct probabilistic behaviour, considering them jointly provides a more complete

```
* * omitted for brevity * *
if (descr->subarray) {
    // handle subarray
    return ret;
}

if ((unsigned int)nd > (unsigned int)
NPY_MAXDIMS) {
    // raise error about dimension size
    return NULL;
}
* * omitted for brevity * *
```

(a) **Vulnerable variant (CWE-120)**: Bounds check occurs *after* subarray handling, allowing unsafe execution paths. *DeepSeek Coder 6.7b metrics*: `Perplexity`=4.98, `LogProb`=−300.22, `Uncertainty`=1.36

```
* * omitted for brevity * *
if ((unsigned int)nd > (unsigned int)
NPY_MAXDIMS) {
    // raise error about dimension size
    return NULL;
}

if (descr->subarray) {
    // handle subarray
    return ret;
}
* * omitted for brevity * *
```

(b) **Patched variant**: The bounds check is performed *before* subarray handling, preventing CWE-120 vulnerabilities. *DeepSeek Coder 6.7b metrics*: `Perplexity`=5.11, `LogProb`=−305.13, `Uncertainty`=1.30

Figure 1: **Metric divergence between secure and insecure code.** Despite being safer, the patched variant (which adds a bounds check to mitigate the buffer-overflow vulnerability *CVE-2021-33430* in NumPy identified by GitHub advisory *GHSA-6p56-wp2h-9hxr*) receives a *higher perplexity* and *lower log probability* than its vulnerable counterpart. This misalignment suggests that DeepSeek-Coder-6.7B assigns greater likelihood and fluency to insecure patterns, counter to the expected behaviour of lower perplexity, higher log probability, and reduced uncertainty for secure code.

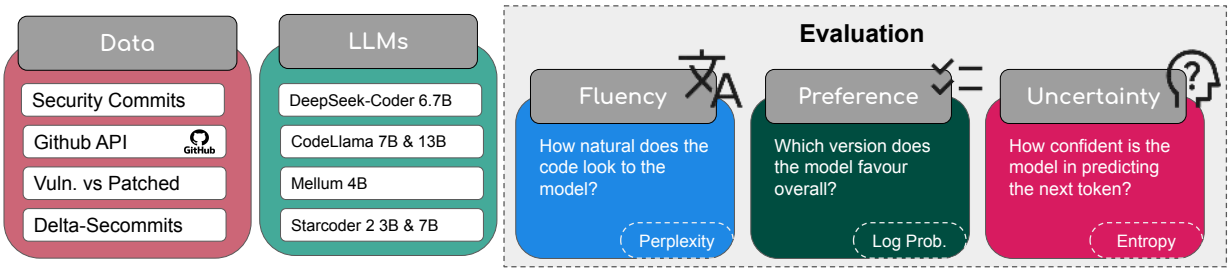


Figure 2: Overview of our methodology for evaluating security misalignment in code-focused LLMs. We curated pairs of vulnerable and patched commits from the Secommits dataset and assessed multiple state-of-the-art LLMs on preference, fluency, and uncertainty.

picture of model alignment. Prior work shows that fluency measured via perplexity is a well-established proxy for the plausibility of the code and text [39], while preference expressed through pairwise log-probability comparisons has been widely used in alignment methods such as Direct Preference Optimisation [40]. At the same time, uncertainty [41] quantified by entropy has been demonstrated as a reliable signal of model confidence and a predictor of unreliable outputs [42]. Since no single metric suffices to fully assess LLM behaviour, combining fluency, preference, and uncertainty allows us to detect subtle forms of security misalignment that might be missed if only one dimension were considered.

A. Dataset Construction

We describe the process used to construct our dataset, including collection, preprocessing, and filtering.

*a) **Data Source**:* Building security vulnerability datasets is challenging due to the prevalence of false positives [43], which can mislead models into learning spurious patterns. To address this, we leverage *Secommits* [44], [45], a curated dataset of vulnerability-fixing commits and metadata drawn from authoritative sources such as the Open Source Vulnerability (OSV) Database and the National Vulnerability Database (NVD). `Secommits` comprising 11205 samples collected between June 1999 and August 2022, covering 278 publicly disclosed vulnerabilities.

*b) **Filtering Pipeline**:* We refined the original dataset to include only single-file commits, thereby reducing label ambiguity and excluding noise from unrelated changes such as tests, documentation, or configuration updates.

1) **Single-file commits.** We retained only commits that modified exactly one file, thereby reducing label ambiguity and excluding unrelated changes (e.g., tests, documentation). This reduced the dataset to 9,029 entries.
2) **Single-commit fixes.** We excluded vulnerabilities requiring multiple commits to resolve, further lowering noise. This yielded 4,315 entries.
3) **Patch retrieval.** Using GitHub's API, we retrieved commit metadata and corresponding patches based on

project names and commit *SHA*s, normalising them into pre- and post-patch versions representing vulnerable and secure states.

4) **Metadata constraints.** Each entry was required to have a unique vulnerability identifier (4,292), a valid CWE identifier (3,562 entries), and a valid severity rating (3,534 entries).
5) **Language filtering.** We restricted entries to files with programming-related extensions (`.java`, `.ts`, `.php`, `.js`, `.cc`, `.py`, `.go`, `.kt`, `.rb`, `.rs`, `.cs`, `.cpp`, `.c`, `.html`, `.xml`), yielding 3,252 entries.
6) **CWE expansion.** A small fraction ($\simeq 1.5\%$) of samples were associated with multiple CWE identifiers. We expanded these cases to ensure an individual contribution per CWE, resulting in 3,308 entries.
7) **Category threshold.** To ensure robust evaluation, we retained only CWE categories with at least 30 samples, reducing the dataset to 2,519 entries.
8) **Valid pre- and post-patch versions.** We filtered out 3 entries that lacked either a pre- or post-patch version, leaving 2,516 entries. Empty strings occur only in two cases in our data: (1) *minified/generated* assets (e.g., .min.js) where the UI/API suppresses the patch (e.g., GHSA-cxm3-v4mv-6mh8 and GHSA-pq37-4c4g-v38c), and (2) one-sided file additions where no prior version exists (CVE-2017-0576). We exclude these pairs.

Stage	Remaining Entries
Original *Secommits*	11,205
Single-file commits	9,029
Single-commit fixes	4,315
Unique vulnerability identifier	4,292
Valid CWE identifier	3,562
Valid severity rating	3,534
Programming language filtering	3,252
CWE expansion	3,308
CWE $\geq$ 30 samples	2,516
Valid pre- and post-patch version	**2,516**

Table I: Dataset filtering pipeline.

*c) **Final Dataset**:* `Delta-Secommits` contains 2,516 vulnerability–fix pairs spanning 25 CWE categories. Each entry consists of a single-file commit with both vulnerable and patched code, an associated CWE identifier, and a severity rating. In total, `Delta-Secommits` includes 2,422 unique vulnerability identifiers. The three most prevalent are **CWE-79**(n=390), **CWE-119** (n=229), and **CWE-125** (n=221), together accounting for roughly 33% of all samples. The remaining +22 categories range from 30 to 180 samples each. Across models, patched snippets are consistently longer than their vulnerable counterparts, with an average token difference ranging from 71 tokens (StarCoder2 3B/7B), 75 (DeepSeek), 79 (CodeLlama 7B/13B) to 136 tokens (Mellum 4B).

B. Selected Models

We select a representative set of LLMs spanning common tasks in software engineering research, including fault localisation [46], program repair [47], vulnerability detection [48], and code completion [49]. Our chosen models come from families of widely studied, publicly available code-focused architectures: `Mellum`, `CodeLlama`, `StarCoder`, and `DeepSeek Coder`. Specifically, we focus on open-source base models (i.e., LLMs in their pre-training state, before any instruction tuning or task-specific fine-tuning) for two key reasons: 1) Our **evaluation requires computing token-level probability distributions**, including perplexity, log probabilities, and entropy, over entire code sequences. This is generally not feasible with proprietary closed-source systems, as they usually provide only sampling APIs and do not expose token-wise probability introspection. 2) By concentrating on base models, we isolate the fundamental probabilistic behaviors learned during pre-training, prior to instruction tuning, RLHF, or task-specific alignment procedures.

The specific models evaluated in this study include:

- `JetBrains/Mellum-4b-base` [50]
- `bigcode/starcoder2-3b` [51]
- `bigcode/starcoder2-7b`
- `meta-llama/CodeLlama-7b-hf` [49]
- `meta-llama/CodeLlama-13b-hf`
- `deepseek-ai/deepseek-coder-6.7b-base` [52]

This selection ensures a diverse coverage of model sizes (ranging from 3B to 13B parameters) and training corpora, while maintaining the transparency required for detailed probabilistic analysis.

C. Probabilistic Metrics

We evaluate each vulnerable–secure pair along three behavioural dimensions that capture how models align with secure coding practices: **fluency**, **preference**, and **uncertainty**. We introduce each metric and illustrate how it captures different facets of security alignment in code. All metrics are computed over the extracted diff context (i.e., the modified hunk with surrounding lines), reflecting the local code region affected by the vulnerability fix. Since both variants share the same contextual framing, the resulting deltas isolate the probabilistic effect of the modified code.

1) Fluency: It reflects whether secure code appears as "natural" to the model as insecure code. A model biased by training data may assign higher fluency to insecure patterns, even when a secure alternative exists. The notion of software naturalness, first articulated by Hindle *et al.* [53], established that source code exhibits strong statistical regularities that can be quantified using language models. Building on this foundation, Ray *et al.* [54] further demonstrated that buggy code is typically less natural than its corrected version, suggesting that lower perplexity should align with more reliable code.

Following this line of work, we measure fluency using perplexity, PPL, a standard token-level metric in NLP and Machine Learning (ML) that reflects the likelihood of a sequence under the model's distribution:

$$\mathrm{PPL}(x) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(x_i \mid x_{<i})\right)$$

where $x = (x_1, \ldots, x_N)$ is a token sequence and $P(x_i \mid x_{<i})$ is the model-assigned probability of token x_i given its prefix. Lower PPL indicates higher fluency. In our setting, this means that if secure code receives lower PPL than its vulnerable counterpart, the model perceives the secure variant as more natural, suggesting stronger alignment towards secure generation. Conversely, higher PPL for secure code indicates that the model is biased toward insecure patterns.

2) Preference: It captures whether a model explicitly favours secure or vulnerable code when both are plausible completions. We compute sequence-level log-probabilities for each variant and compare them using the Bradley–Terry (BT) [40], [55] formulation:

$$\log P(x) = \sum_{i=1}^{N} \log P(x_i \mid x_{<i})$$

$$\mathcal{L}_{\text{BT}} = \log\big(1 + \exp\big(-\beta(\log P_{\text{secure}} - \log P_{\text{vuln}})\big)\big)$$

where β is a temperature parameter (default $\beta = 1$). Lower $\mathcal{L}_{\text{BT}}$ corresponds to a stronger preference for the secure completion. In our context, this means the model assigns greater probability to secure code over its vulnerable counterpart, indicating alignment towards secure generation. Conversely, higher values suggest the model either favours or does not sufficiently penalise insecure variants.

In practice, rather than directly computing $\log\big(1+\exp(\delta)\big)$, with $\delta = -\beta(\log P_{\text{secure}} - \log P_{\text{vuln}})$, we use the *softplus* function. *softplus* is numerically more stable, especially when δ takes large positive values. It provides a smooth approximation of the rectifier and ensures stable gradients during optimisation, avoiding overflow or underflow for extreme δ:

$$\text{softplus}(\delta) = \max(0, \delta) + \log\big(1 + \exp(-|\delta|)\big).$$

Preference is defined as the cumulative sum of log probabilities of each token given the preceding context. Unlike perplexity, there is no normalization term, since token sequences can be of arbitrary length. We note that a normalised version of Preference closely reduces to the perplexity measure. Precisely because it is length-invariant, it fails to capture the systematic penalty incurred by longer, security-relevant patches. For this reason, we retain the cumulative metric, even though it is sensitive to length.

3) Uncertainty: It is critical for security-sensitive settings: if a model is less confident about secure patterns, it may revert to unsafe defaults. We measure predictive uncertainty via average token-level Shannon entropy [56]:

$$\text{Entropy}(x) = \frac{1}{N} \sum_{i=1}^{N} \left[-\sum_{v \in V} P(v \mid x_{<i}) \log P(v \mid x_{<i}) \right]$$

where V is the model vocabulary. Lower entropy indicates greater confidence in the predicted next token. Lower entropy on secure code suggests the model is confident in generating secure patterns, whereas higher entropy indicates uncertainty that may cause the model to revert to insecure defaults.

D. Security Misalignment

We view alignment as a bundle of three probabilistic behaviours (ref. Section IV-C) that together capture how LLMs probabilistically **evaluate** secure versus vulnerable code. When secure variants are at least as fluent, preferred, and confident as their vulnerable counterparts, we consider the model *aligned*. When vulnerable variants are favored along one or more dimensions, we observe *security misalignment*. For each pair i, consisting of a secure variant i_{secure} and its vulnerable counterpart i_{vuln}, we calculate following Δs:

$$\Delta^i_{\text{PPL}} = \text{PPL}(i_{\text{vuln}}) - \text{PPL}(i_{\text{secure}})$$
$$\Delta^i_{\text{LogProb}} = \log P(i_{\text{secure}}) - \log P(i_{\text{vuln}})$$
$$\Delta^i_{\text{Entropy}} = \text{Entropy}(i_{\text{vuln}}) - \text{Entropy}(i_{\text{secure}})$$

Positive Δ values indicate that the secure variant is favoured, while negative values indicate that the vulnerable variant is favoured. We define *security misalignment* as the case where one or more $\Delta < 0$, and *full alignment* as the case where all three $\Delta \geq 0$ (see Table II).

Table II: Interpretation of Δ values for alignment.

Condition	Interpretation
$\Delta \geq 0$ (for a given metric)	Secure variant is favoured
$\Delta < 0$ (for a given metric)	Vulnerable variant is favoured
All $\Delta \geq 0$	*Full alignment* (secure $\geq$ vulnerable on all dimensions)
One or more $\Delta < 0$	*Security misalignment* (secure $<$ vuln. on at least one dimension)

By jointly analysing Δ_{PPL}, Δ_{LogProb}, and Δ_{Entropy}, we go beyond surface-level fluency to capture how models both prefer and trust secure code. This integrated view is essential for assessing security alignment in LLM-based code generation.

E. Statistical Testing

To assess the significance of differences in performance, we apply the Wilcoxon signed-rank test [57], grouping results by CWE category and model. For each pair, we compute the test on the distribution of PPL differences. A threshold of $p < 0.05$ is used to determine statistical significance, and non-significant cells are explicitly marked in the visualization.

V. Results

We empirically evaluate six code-focused LLMs on secure vs. insecure code using the three probabilistic metrics from Section IV: fluency, preference, and uncertainty. Results are organised by research question.

RQ1: Our multi-metric framework shows that token-level probability distributions provide a principled lens for quantifying security alignment. Fluency, measured via perplexity, captures how naturally models generate code but does not inherently encode security preferences. Preference-based metrics, derived from pairwise probability comparisons between secure and vulnerable implementations, provide more direct evidence

of alignment with secure practices. Uncertainty metrics reveal cases where the model is indecisive. Because LLMs are trained on uncurated corpora that contain bugs and vulnerabilities [9], [24], these metrics largely reflect the training distribution, often reproducing statistically prevalent insecure patterns.

The partial decoupling of these metrics shows that strong fluency or low uncertainty alone cannot guarantee security alignment. Thus, security-aware operationalization requires explicitly integrating preference signals, rather than relying on general-purpose fluency or uncertainty measures.

a) Metric Independence Analysis: To ensure that our three evaluation metrics, preference, perplexity, and uncertainty differences, capture complementary aspects of model behaviour, we conducted two orthogonal analyses: pairwise correlation analysis and principal component analysis (PCA).

Table 3a presents the correlation matrix across all evaluation metrics. We observe a moderately strong correlation between perplexity and uncertainty differences ($r = 0.679$). In contrast, preference shows weak negative correlations with both perplexity ($r = -0.130$) and uncertainty ($r = -0.191$). This suggests that preference varies largely independently of perplexity and uncertainty, indicating that these metrics capture distinct aspects of model behaviour.

PCA results (Table 3b) further support this:

- **PC1** is a combined axis primarily loading on preference and perplexity (loadings: 0.6957 and 0.6970), suggesting a shared variance that partially mixes these two aspects.
- **PC2** is dominated by uncertainty (loading: 0.9794), representing a nearly pure axis of uncertainty differences.
- **PC3** captures a contrast between preference and perplexity (loadings: -0.6897 and 0.7167), highlighting a residual dimension where these two metrics diverge.

The three principal components explain approximately 100% of the total variance, with PC1 alone accounting for about 63.1%, PC2 for 25.0%, and PC3 for 11.9%. This underscores that each metric contributes with unique information, with PC1 capturing the dominant shared variance, while PC2 and PC3 capture more specialised variation across metrics.

*b) **Answer to RQ1: Operationalising Security Alignment.**:* Security alignment can be operationalised by comparing secure and vulnerable code variants along three probabilistic metrics: fluency (perplexity), preference (log-probability), and confidence (entropy). A model is fully aligned when secure code is at least as fluent, preferred, and confident as its insecure counterpart; misalignment occurs whenever one or more of these conditions is violated. Our results confirm that these metrics are not interchangeable but complementary: perplexity, log-probability, and entropy each capture distinct aspects of model behaviour. This validates our multi-dimensional operationalisation of security alignment. Importantly, cases of partial alignment, where only one or two metrics favour secure code, are not noise but a meaningful diagnostic of model bias.

Table III illustrates how different combinations of positive and negative Δ values correspond to distinct patterns of misalignment. Crucially, these patterns map to three complementary levels of intervention. *Data-level* interventions address cases where secure variants are preferred or confident but lack fluency, by enriching exposure to idiomatic secure code in training corpora (e.g., CWE-specific examples). *Model-level* interventions are required when the probability landscape itself is biased toward insecure code, calling for techniques such as contrastive fine-tuning, RLHF/DPO, or adversarial training to rebalance preferences. *System-level* interventions complement the first two by providing runtime safeguards, such as IDE linting, CI guardrails, or presenting multiple candidate completions, when models still default to insecure suggestions. Together, the table and these categories demonstrate that security alignment is not a binary property but a spectrum of behaviours. Partial alignment patterns serve as actionable diagnostics, guiding both model improvement and tool design for secure LLM-assisted development.

	Preferred	$\text{PPL}_{>0}$	$\text{Uncert}_{>0}$
Preferred	1.000	-0.1178	-0.1801
$\text{PPL}_{>0}$	-0.1178	1.000	0.6746
$\text{Uncert}_{>0}$	-0.1801	0.6746	1.000

(a)

Metric	**PC1**	**PC2**	**PC3**
Alignment (Δ_{LogProb})	0.6970	0.1963	-0.6897
Perplexity (Δ_{PPL})	0.6957	0.0479	0.7167
Uncertainty (Δ_{Entropy})	-0.1738	0.9794	0.1032

(b)

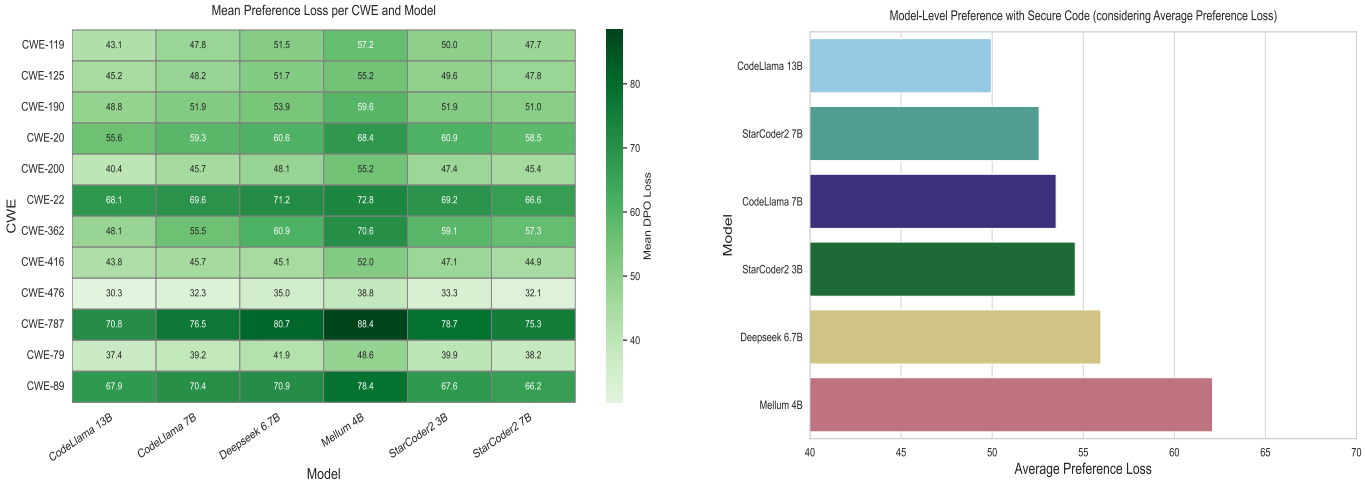
Figure 3: (a) Correlation matrix of evaluation metrics. (b) Principal component loadings across alignment, perplexity, and uncertainty. Each loading reflects the weight (or contribution) of a metric to the corresponding principal component: higher absolute values indicate stronger influence on that axis, while the sign denotes the direction of the relationship. PC1 combines alignment and perplexity as a shared variance axis, PC2 is almost entirely defined by uncertainty, and PC3 contrasts alignment with perplexity. These three components explain nearly all variance, underscoring that the metrics capture complementary aspects of model behaviour.

RQ2: We have observed that in $\sim 92\%$ of the cases, LLMs exhibit security misalignment across models and CWEs. While models exhibit competence on certain classes of vulnerabilities, they consistently underperform on memory-safety issues, suggesting structural blind spots in their learned distributions. Even when fluent and confident, models rarely show an explicit probabilistic preference for secure code—emerging in only $\sim$8% of evaluated cases. This indicates that current base LLMs are not inherently security-aligned, and instead often privilege fluency over security. Addressing this gap requires CWE-targeted fine-tuning or the incorporation of security objectives directly into training, ensuring that probabilistic preferences align with secure coding practices in a reliable and systematic manner.

*c) **Per-CWE Analysis**:* For each CWE, we compute Δ_{PPL}, Δ_{LogProb}, $\mathcal{L}_{\text{BT}}$, and Δ_{Entropy}. Figure 4 provides a detailed

Table III: Interpretation of different alignment patterns across fluency, preference, and confidence for secure code. Positive Δ (+) means the secure variant is favoured; negative Δ (-) means the vulnerable variant is favoured.

Δ_{PPL}	Δ_{LogProb}	Δ_{Entropy}	**Interpretation**	**Action/Implication**
+	+	+	**Full alignment:** Secure code is more fluent, preferred, and confidently generated.	Ideal. Maintain alignment in future releases.
-	-	-	**Full misalignment:** Insecure code dominates across all metrics.	Highest risk. Requires *model-level* rebalancing (e.g., contrastive fine-tuning).
+	-	-	Secure code is more fluent but not preferred or confidently generated.	Needs *model-level* adjustment (preference reweighting) and *system-level* support (multiple completions shown).
-	+	-	Secure code is explicitly preferred but less fluent and less confidently predicted.	Calls for *data-level* enrichment (idiomatic secure examples).
-	-	+	Model is confident in insecure variants despite secure code being less natural/preferred.	Requires *model-level* preference rebalancing, plus *system-level* guardrails (linting, IDE warnings).
+	+	-	Secure code is fluent and preferred, but with low certainty.	Suggests *model-level* fine-tuning to reduce entropy.
+	-	+	Secure code is fluent and confident, but preference still skews toward insecure.	Needs *model-level* correction (contrastive training).
-	+	+	Secure code is preferred and confident, but not fluent.	Suggests *data-level* augmentation with idiomatic secure corpora.



(a) Heatmap showing the average Preference loss across the 12 most prominent CWEs for each model. Darker shades indicate stronger misalignment with secure coding practices.

(b) Average Preference loss per model. Lower values indicate a stronger preference for secure completions.

Figure 4: Model alignment with secure coding practices across CWE categories. While some CWEs like injection types show strong secure preference, others, particularly memory corruption bugs, exhibit high misalignment.

view of model misalignment with secure coding practices based on Bradley-Terry loss. Subfigure (a) shows the average Preference loss per CWE and model, revealing that alignment is highly vulnerability-specific. Memory safety issues, such as CWE-787 (Out-of-bounds Write) and CWE-125 (Out-of-bounds Read), consistently produce the highest Preference losses, indicating that models frequently prefer insecure variants. In contrast, injection-related CWEs (e.g., CWE-89, CWE-94) yield lower Preference losses, suggesting more reliable secure preferences in those domains. Subfigure (b) summarises model-level alignment, with StarCoder2 7B exhibiting the lowest overall Preference loss, while Mellum 4B shows the weakest alignment. These results emphasise the need for CWE-aware evaluation when assessing security alignment in code generation models. Interestingly, we also observe that larger architectures (e.g., Starcoder2 3B $\rightarrow$ 7B and CodeLlama 7B $\rightarrow$ 13B) achieve a lower average Preference loss, as shown in Subfigure (b). This trend may reflect the emergent properties of large language models [58], where increased model capacity leads to better alignment with user preferences.

*d) **Overall Model Trends**:* Table IV summarises results across models. While secure code often exhibits lower perplexity and entropy, explicit preferences still overwhelmingly favour insecure variants. Here, the preference percentage reflects the proportion of examples where the secure (patched) code is preferred under the Bradley-Terry formulation. We observe that while models often exhibit lower perplexity (improved fluency) and reduced entropy (greater confidence) on secure code, they still prefer insecure variants $\sim 85\%$ of the time on average. Our analysis reveals that the $\sim 92\%$ aggregate misalignment obscures critical CWE-specific patterns. As shown in Figure 5, the degree of misalignment varies

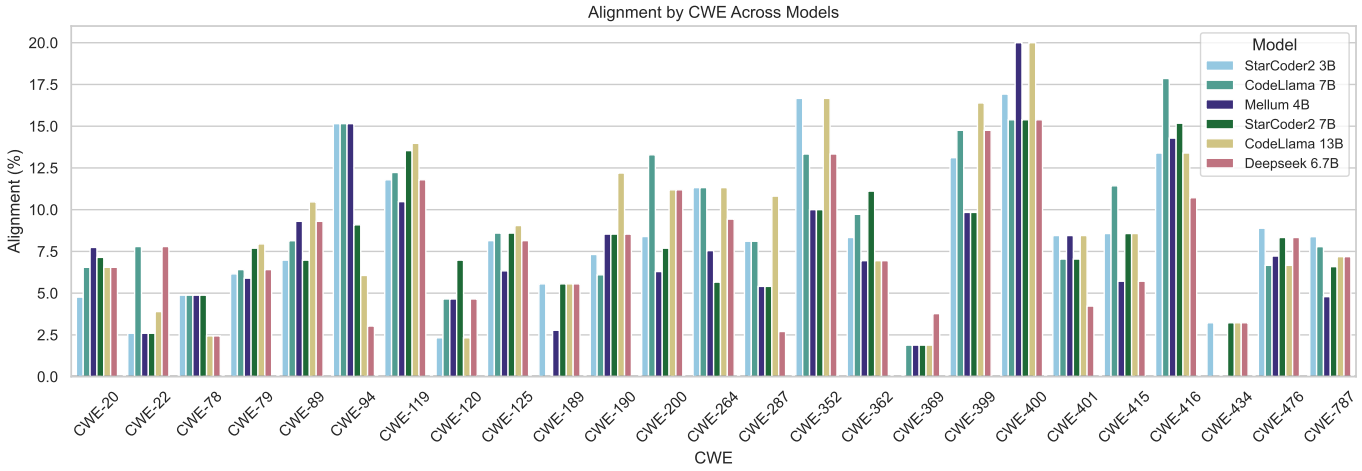


Figure 5: Security alignment per model across CWES

Table IV: Summary of model alignment with secure code. Models show lower Preference for secure code despite high Fluency and Certainty. Preference (Pref %) is based on positive Δ_{LogProb}, Fluency is based on positive Δ_{PPL} and Certainty (Cert %) is based on positive Δ_{Entropy}. Additionally, Alignment (Align %) combines preference with positive Δ_{PPL}, Δ_{Entropy} and Δ_{LogProb}.

Model	Cert. (%)	Fluency (%)	Pref (%)	Align (%)
CodeLlama-7B	75.60	75.36	16.02	8.86
CodeLlama-13B	75.28	75.20	16.34	9.18
StarCoder2-3B	76.35	74.56	15.46	8.23
StarCoder2-7B	76.23	74.44	16.06	8.47
Mellum-4B	72.42	72.54	15.26	7.51
DeepSeek-6.7B	75.60	75.91	14.98	8.15

substantially across CWE categories, revealing critical patterns in model behaviour. Most notably, CWE-369 exhibits near-total misalignment, with correct security preferences appearing in merely $\sim 2.8\%$ of evaluations for StarCoder models and DeepSeek-Coder, while CodeLlama variants failed completely to align with secure implementations. In contrast, CWE-400 shows modest improvement with $\sim 20\%$ correct alignment. This striking variance across vulnerability types highlights how current models develop specialised but incomplete security competencies, excelling in certain vulnerability classes while catastrophically failing in others. The consistent preference for insecure code across most categories underscores fundamental limitations in how LLMs internalise security principles during training.

Uncertainty analysis further reveals that models are not consistently more confident in secure code. In some CWE categories, patched variants even show higher entropy, indicating hesitancy despite improved fluency. Quantitatively, $\sim 60.4\% \pm 1.0$ of examples exhibit positive ΔPPL and ΔEntropy with negative ΔLogProb; $\sim 13.7\% \pm 1.0$ have all three metrics negative; $\sim 8.4\% \pm 0.58$ show all three metrics positive, and the remaining combinations each occur in less than 5.8% of cases.

We acknowledge that our union-style definition of misalignment, where any single negative delta suffices, yields an inclusive threshold. The 92% figure is driven by the preference axis: in 60% of cases, models assign higher fluency and confidence to secure code yet still prefer the insecure variant under cumulative log-probability. We retain this definition because preference determines which completion is emitted during generation. Thus, even when a model recognises secure code as fluent and expresses confidence in it, assigning higher cumulative probability to the insecure variant implies it will generate the insecure version under greedy decoding. Table III therefore provides an axis-specific breakdown for readers seeking a more graded view of alignment.

*e) **Statistical Testing**:* Across the 150 CWE–model comparisons evaluated using the Wilcoxon signed-rank test for the PPL analysis, 134 exhibit statistically significant differences ($p < 0.05$), indicating that the majority of observed effects are robust, while a smaller portion of comparisons do not reach significance. For the uncertainty analysis, 130 out of the same 150 comparisons yield statistically significant differences ($p < 0.05$), again demonstrating consistent divergence across most models and CWE categories. Finally, when applying the Wilcoxon signed-rank test to the BT loss, all 150 comparisons produce statistically significant differences ($p < 0.05$). This complete consistency reinforces the robustness of the findings across different evaluation metrics and tasks.

*f) **Top 5 Misaligned CWEs**:* To illustrate security misalignment in greater depth, we focus on the five CWE categories where CodeLlama-13B shows the weakest preference for secure code. These represent the model's most critical failure modes, in which insecure variants are systematically favoured. Table V summarises the corresponding metrics. All five categories exhibit high average Bradley–Terry losses, indicating persistent difficulty in internalising a preference for secure code. **CWE-264** shows the highest Bradley-Terry loss (74.16) and the largest Δ_{Uncert} (0.1698), yet achieves a prefer-

ence rate of only $\sim 14\%$. Similarly, **CWE-787** (Out-of-bounds Write) combines high loss (70.76) with elevated Δ_{Uncert} (0.1138) but an even lower preference rate of $\sim 12\%$. These cases expose a critical disconnect: increased confidence in secure completions does not translate into stronger preference for them. **CWE-89** (SQL Injection) is notable for the highest Δ_{PPL} (2.02), suggesting secure completions are substantially more fluent, yet the preference rate remains modest ($\sim 14\%$). Overall, these results reaffirm that improvements in fluency (Δ_{PPL}) and confidence (Δ_{Uncert}) alone are insufficient to ensure that CodeLlama prioritises secure code. Even when considering only fluency and confidence—excluding preference/log-probability—the proportion of aligned cases remains below 70% across all evaluated LLMs, leaving a persistent $\sim 30\%$ misalignment. This confirms that fluency and confidence are not reliable proxies for security alignment.

Table V: Performance metrics by CWE category with at least 30 samples for `CodeLlama-13b-hf`. *Avg BT Loss* and *Avg* Δ_{PPL} denote the mean Bradley-Terry loss and perplexity differences across instances. *Preference Rate* is the fraction of cases where the secure variant is explicitly preferred.

CWE	Avg BT Loss	Avg Δ_{PPL}	Avg Δ_{Uncert}	Pref. Rate
CWE-264	74.1619	1.2422	0.0850	0.1698
CWE-787	70.7559	1.0981	0.1188	0.1138
CWE-22	68.0552	0.7603	0.0692	0.1038
CWE-89	67.9255	2.0177	0.1034	0.1395
CWE-434	57.8781	1.5973	0.0967	0.1290

Length control. Although the preference metric is mechanically sensitive to sequence length, length differences do not explain the effect. For each pair, we computed the Pearson correlation between the token-count difference (patched minus vulnerable) and the binary alignment outcome (1 if the model ranked the secure completion higher, 0 otherwise). For CodeLlama-7B, this correlation is negligible ($r = -0.078$, $R^2 < 1\%$), and logistic regression using token-count difference as the sole predictor shows no significant effect ($p = 0.595$). Results are similarly negligible across all models.

Class-distribution control. Micro- and macro-averaged alignment rates differ by at most 0.57% across models, with near-identical rankings, indicating that no single CWE category drives the results.

Duplicate control. Removing duplicate $(\texttt{vuln\_id}, \texttt{model})$ pairs changes alignment rates by at most 0.10%, confirming robustness to repeated vulnerabilities.

Effect Magnitude. To quantify how strong the observed preferences are, we computed rank-biserial correlations ($r = Z/\sqrt{N}$) for each comparison. Across the 430 significant comparisons, the median effect size was $r = 0.76$, , which is considered large. The effect was strongest for Δ_{PPL} ($r = 0.83$) and Δ_{LogProb} ($r = 0.86$), indicating that LLMs assign substantially higher probability to insecure code variants. The effect for Δ_{Entropy} was moderate ($r = 0.54$), suggesting that models are also more confident, less uncertain, when generating insecure code. To account for the large number of tests (450), we applied Benjamini-Hochberg correction; all 430 significant results held at a false discovery rate of 0.05, confirming that the observed patterns are robust to multiplicity.

VI. Discussion

Our evaluation exposes a **systemic security misalignment** in LLM-assisted code generation: while LLMs can recognize secure code as syntactically fluent, they **probabilistically favor insecure variants** in $\sim 92\%$ of cases. This bias is not neutral: it prioritizes fluency and statistical patterns over security. For the testing and verification community, this misalignment raises concerns about the reliability of AI-assisted tools, as insecure code may be generated with high confidence and integrated into software systems without detection.

Implications for Evaluation and Tooling. Current benchmarks for LLMs [8]–[10], [15]–[25] largely focus on functional correctness, fluency, or style. Our results show these criteria are insufficient: insecure code can be both fluent and highly probable even when secure alternatives exist. Future testing and evaluation must treat **security alignment as a first-class concern**, extending the current benchmarking culture, which often emphasizes correctness and performance over security preference. This calls for new datasets, tasks, and metrics that assess whether secure variants are preferred over insecure ones during generation and repair tasks.

Developer and Industry Impact. In practice, misalignment affects everyday workflows, including pair programming, pull requests, test generation, and CI/CD pipelines. Each insecure suggestion increases the burden of manual review and penetration testing, eroding productivity while raising the risk of vulnerabilities entering production systems. At scale, this risk extends across industry: widely deployed assistants such as Copilot and ChatGPT [9], [24], [52], [59]–[64] may propagate insecure defaults across software ecosystems, amplifying attack surfaces and threatening supply-chain security.

Addressing Misalignment. It is important to frame our findings not as an architectural failing of LLMs, but as a consequence of their training data. LLMs amplify the insecure patterns they were trained on, posing a systemic risk. Mitigating these risks requires shifting probabilistic preferences toward secure defaults. Techniques such as contrastive training with secure exemplars, fine-tuning on curated datasets like `Delta-Secommits`, or hybrid pipelines combining LLMs with static analysis could shift probability landscapes toward secure defaults. Prompt-based hardening, such as the security-aware system prompts studied by He and Vechev [65], offers another mitigation path by conditioning model outputs toward secure generation. Our study measures the underlying probabilistic bias that such prompts must counteract. A model with a strong baseline preference for insecure code requires more aggressive prompt intervention than one that is closer to neutral. In this sense, our results quantify the magnitude of the problem that prompt-based defences need to overcome, rather than conflicting with their effectiveness. The priority is not just building new tools, but rethinking how both the underlying

AI models and the developer-facing tools built on top of them are trained, evaluated, and tested before deployment.

Limitations and Future Research. Our framework targets the model's probabilistic behaviour, the internal biases that precede and inform generation. Generation behaviour introduces additional variables (decoding strategy, temperature, prompt design, stopping criteria) that interact with these biases in complex ways. Characterising that interaction across decoding configurations is a natural next step that warrants its own dedicated study. Systematic biases at the probability level propagate into generation under standard decoding unless explicitly counteracted. More broadly, addressing misalignment requires embedding security directly into model objectives [66], augmenting preference optimisation, applying adversarial training [65], and designing hybrid objectives that integrate *fluency*, *preference*, *uncertainty*, and program safety. Advances in mechanistic interpretability [67]–[70] and representation engineering [71]–[73] further suggest that targeted interventions in representation space, where models encode latent secure and insecure patterns, could support more trustworthy alignment.

VII. Threats to Validity

Our work is subject to several potential limitations that may influence the interpretation and generalisation of results. Below, we discuss threats across internal, external, construct, and conclusion validity, highlighting the factors that may compromise robustness and suggesting ways to mitigate them.

Internal Validity. Our evaluation may be influenced by confounding factors. In particular, 4-bit quantisation introduces approximation errors in model weights, which may distort probability estimates, entropy, and perplexity. However, our methodology relies on paired comparisons: each delta is computed between a vulnerable and patched variant scored by the same quantised model under identical conditions. While quantisation adds noise to individual probability estimates, this noise affects both variants equally, and there is no known mechanism by which it would inflate scores for vulnerable code over patched code. We additionally observe consistent trends across six models with different architectures, parameter counts, and vocabularies, which would be unlikely if quantisation artefacts were driving the results. Smaller effects may partially reflect quantisation noise, and replication at higher precision remains desirable for future work.

External Validity. The generalisability of our results is limited by the scope of the study. We focus on a limited set of LLMs and programming languages, and results may not extend to other model families, tokenisation strategies, or development contexts. Real-world workflows are heterogeneous and may introduce factors not captured in our study. Replicating the analysis with additional datasets, languages, and architectures is necessary to assess broader applicability.

Construct Validity. Our conclusions rely on metrics such as perplexity, log probability, and entropy as proxies for security alignment. While informative for fluency and confidence, these metrics are indirect and may overlook subtle vulnerabilities. Moreover, they are tightly coupled to model-specific tokenisers. For instance, CodeLlama and StarCoder tokenisation differ substantially. CodeLlama's tokenisation shows a fine-grained subword splitting (e.g., `SHA244` → `S`, `HA`, `2`, `4`, `4` and `digest` → `dig`, `est`), comparing to StarCoder's tokenisation with more semantic splits (e.g., `SHA244` → `SHA`, `2`, `4`, `4` and preserving whole words like `digest`). Cross-model comparisons should be interpreted with caution. However, our main conclusions do not depend on such comparisons: the misalignment finding holds within each model individually, with all six models exhibiting alignment rates below 10%. In addition, the *preference* metric is sensitive to patch length, potentially conflating sequence size with alignment signals. These factors threaten construct validity through both metric indirectness and implementation-specific dependencies.

Conclusion Validity. Probabilistic metrics are inherently noisy, and comparisons should therefore be interpreted with caution. Although we observe consistent trends across models and vulnerabilities, our analysis does not exhaustively control for all sources of statistical uncertainty. Future work should incorporate more statistical validation and larger, more diverse datasets to strengthen confidence in the conclusions.

VIII. Conclusion

This paper introduces a **probabilistic testing framework** to assess security alignment in code LLMs, revealing that state-of-the-art base models exhibit a systematic preference for insecure patterns across a wide range of vulnerability categories. By jointly analyzing fluency, preference, and uncertainty, we demonstrate that secure code variants are rarely favored across all dimensions, with full alignment in only **8%** of cases.

Our contributions include probabilistic evaluation metrics that expose security misalignment at a deeper level than traditional correctness-based benchmarks. The results highlight a structural bias in current models: even when LLMs produce fluent and confident code, their underlying probabilistic preferences often favor insecure variants. This misalignment poses a direct challenge to software testers, QA engineers, and verification tools that increasingly rely on LLMs for code generation, test synthesis, and automated repair.

By embedding security into the **probabilistic foundations** of code generation, we can narrow the gap between fluent AI assistance and the guarantees expected by software testing and verification. We call for a paradigm shift in how the testing community approaches LLM-integrated development—away from treating security as a downstream correction, and toward **proactive, model-internal verification** of security alignment.

Acknowledgments

The authors would like to thank Leyi Cui for providing constructive feedback on this paper. Rui Melo is funded by Fundação para a Ciência e Tecnologia (FCT) through the CMU Portugal Dual PhD Program. This work is also supported by the SALVE project (2024.14936.PEX) and HPC (2025.08039.CPCA).

REFERENCES

[1] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 21–29. [Online]. Available: https://doi.org/10.1145/3520312.3534864

[2] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, and Jiang, "Github copilot ai pair programmer: Asset or liability?" 2023. [Online]. Available: https://arxiv.org/abs/2206.15331

[3] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2205–2222. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

[4] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" in *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2023, pp. 2445–2451.

[5] M. Gupta, C. Akiri, K. Aryal, E. Parker, and L. Praharaj, "From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy," *IEEE Access*, vol. 11, pp. 80 218–80 245, 2023.

[6] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *Empirical Software Engineering*, vol. 28, no. 6, p. 129, Sep 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10380-1

[7] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," 2024. [Online]. Available: https://arxiv.org/abs/2308.04838

[8] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2785–2799. [Online]. Available: https://doi.org/10.1145/3576915.3623157

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[10] G. Dolcetti, V. Arceri, E. Iotti, S. Maffeis, A. Cortesi, and E. Zaffanella, "Helping llms improve code generation using feedback from testing and static analysis," 2025. [Online]. Available: https://arxiv.org/abs/2412.14841

[11] M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "Promsec: Prompt optimization for secure generation of functional source code with large language models (llms)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2266–2280. [Online]. Available: https://doi.org/10.1145/3658644.3690298

[12] K. Ton, N. Nguyen, M. Nazzal, A. Khreishah, C. Borcea, N. Phan, R. Jin, I. Khalil, and Y. Shen, "Sgcode: A flexible prompt-optimizing system for secure generation of code," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 5078–5080.

[13] M. L. Siddiq, B. Casey, and J. C. S. Santos, "Franc: A lightweight framework for high-quality code generation," 2024. [Online]. Available: https://arxiv.org/abs/2307.08220

[14] N. T. Islam, J. Khoury, A. Seong, M. B. Karkevandi, G. D. L. T. Parra, E. Bou-Harb, and P. Najafirad, "Llm-powered code vulnerability repair with reinforcement learning and semantic reward," 2024. [Online]. Available: https://arxiv.org/abs/2401.03374

[15] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "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)*, 2025, pp. 33–40.

[16] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[17] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[18] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[19] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," *arXiv preprint arXiv:2408.06450*, 2024.

[20] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[21] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[22] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[23] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia *et al.*, "Recode: Robustness evaluation of code generation models," *arXiv preprint arXiv:2212.10264*, 2022.

[24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[25] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[26] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[27] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[28] L. Fang, Y. Wang, Z. Liu, C. Zhang, S. Jegelka, J. Gao, B. Ding, and Y. Wang, "What is wrong with perplexity for long-context language modeling?" 2025. [Online]. Available: https://arxiv.org/abs/2410.23771

[29] D. Ulmer, J. Frellsen, and C. Hardmeier, "Exploring predictive uncertainty and calibration in nlp: A study on the impact of method & data scarcity," *arXiv preprint arXiv:2210.15452*, 2022.

[30] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," *arXiv preprint arXiv:1909.08593*, 2019.

[31] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, "Learning to summarize with human feedback," *Advances in neural information processing systems*, vol. 33, pp. 3008–3021, 2020.

[32] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, D. Gabi, D. Song, F. Ahmad, C. Aschermann, L. Fontana *et al.*, "Purple llama cyberseceval: A secure coding benchmark for language models," *arXiv preprint arXiv:2312.04724*, 2023.

[33] S. Wan, C. Nikolaidis, D. Song, D. Molnar, J. Crnkovich, J. Grace, M. Bhatt, S. Chennabasappa, S. Whitman, S. Ding *et al.*, "Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models," *arXiv preprint arXiv:2408.01605*, 2024.

[34] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2024, pp. 684–709.

[35] J. Wang, L. Cao, X. Luo, Z. Zhou, J. Xie, A. Jatowt, and Y. Cai, "Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation," *arXiv preprint arXiv:2310.16263*, 2023.

[36] A. Rydén, E. Näslund, E. M. Schiller, and M. Almgren, "Llmseccode: Evaluating large language models for secure coding," in *Cyber Security, Cryptology, and Machine Learning*, S. Dolev, M. Elhadad, M. Kutyłowski, and G. Persiano, Eds. Cham: Springer Nature Switzerland, 2025, pp. 100–118.

[37] M. L. Siddiq, J. Santos, S. Devareddy, and A. Muller, "Sallm: Security assessment of generated code," *arXiv preprint arXiv:2311.00889*, 2023.

[38] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "How secure is ai-generated code: a large-scale comparison of large language models," *Empirical Software Engineering*, vol. 30, no. 2, p. 47, Dec 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10590-1

[39] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *Proceedings - International Conference on Software Engineering*, pp. 837–847, 06 2012.

[40] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in neural information processing systems*, vol. 36, pp. 53 728–53 741, 2023.

[41] Z. Xia, J. Xu, Y. Zhang, and H. Liu, "A survey of uncertainty estimation methods on large language models," *arXiv preprint arXiv:2503.00172*, 2025.

[42] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal, "Detecting hallucinations in large language models using semantic entropy," *Nature*, vol. 630, pp. 625–630, 06 2024.

[43] A. Bessey, K. Block, B. Chelf, A. Chou, S. Hallem, C. Henri-Gros, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," in *Communications of the ACM*, vol. 53, no. 2. ACM, 2010, pp. 66–75.

[44] S. Reis, R. Abreu, and C. Pasareanu, "Towards security commit message standardization," in *Proceedings of the 22nd International Conference on Mining Software Repositories (MSR)*. Ottawa, Canada: ACM, Apr. 2025.

[45] S. Reis and R. Abreu, "A ground-truth dataset of real security patches," *arXiv preprint arXiv:2110.09635*, 2021.

[46] A. Z. H. Yang, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Large language models for test-free fault localization," 2023. [Online]. Available: https://arxiv.org/abs/2310.01726

[47] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," 2024. [Online]. Available: https://arxiv.org/abs/2312.15698

[48] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," 2024. [Online]. Available: https://arxiv.org/abs/2401.15468

[49] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023. [Online]. Available: https://arxiv.org/abs/2308.12950

[50] N. Pavlichenko, I. Nazarov, I. Dolgov, E. Garanina, K. Lasocki, J. Reshetnikova, S. Boitsov, I. Bondyrev, D. Karaeva, M. Sheptyakov, D. Ustalov, A. Mukhin, S. Proshev, N. Abramov, O. Kolomyttseva, K. Lysaniuk, I. Zavidnyi, A. Semenkin, V. Tankov, and U. Sazanovich, "Mellum-4b-base," 2025.

[51] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[52] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[53] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.

[54] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 428–439. [Online]. Available: https://doi.org/10.1145/2884781.2884848

[55] R. A. Bradley and M. E. Terry, "Rank analysis of incomplete block designs: I. the method of paired comparisons," *Biometrika*, vol. 39, no. 3/4, pp. 324–345, 1952.

[56] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[57] D. Rey and M. Neuhäuser, *Wilcoxon-Signed-Rank Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1658–1659. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_616

[58] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[59] J. D. Weisz, S. V. Kumar, M. Muller, K.-E. Browne, A. Goldberg, K. E. Heintze, and S. Bajpai, "Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise," in *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3706599.3706670

[60] GitHub, "Github copilot: Your ai pair programmer," 2023, accessed: 2025-03-05. [Online]. Available: https://github.com/features/copilot

[61] Tabnine, "Ai code assistant," https://www.tabnine.com/, accessed: 2025-03-05.

[62] M. Jamdade and Y. Liu, "A pilot study on secure code generation with chatgpt for web applications," in *Proceedings of the 2024 ACM Southeast Conference*, 2024, pp. 229–234.

[63] X. Zhang and Others, "Can chatgpt fix your code? evaluating zero-shot code repair with llms," in *Proceedings of the 2023 Conference on Neural Information Processing Systems (NeurIPS)*. NeurIPS, 2023.

[64] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can chatgpt support developers? an empirical evaluation of large language models for code generation," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 167–171. [Online]. Available: https://doi.org/10.1145/3643991.3645074

[65] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1865–1879. [Online]. Available: https://doi.org/10.1145/3576915.3623175

[66] R. Melo, *Securing Language Models Against Vulnerability Encoding*. New York, NY, USA: Association for Computing Machinery, 2025, p. 1277–1278. [Online]. Available: https://doi.org/10.1145/3696630.3731466

[67] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey, "Sparse autoencoders find highly interpretable features in language models," *arXiv preprint arXiv:2309.08600*, 2023.

[68] T. Bricken, A. Templeton, J. Batson, B. Chen, and A. Jermyn, "Towards monosemanticity: Decomposing language models with dictionary learning," *Transformer Circuits Thread*, 2023. [Online]. Available: https://transformer-circuits.pub/2023/monosemantic-features/index.html

[69] L. Gao, T. d. la Tour, H. Tillman, G. Goh, and R. Troll, "Scaling and evaluating sparse autoencoders," *arXiv preprint*, 2024.

[70] R. Melo, C. Mamede, A. Catarino, R. Abreu, and H. L. Cardoso, "Are sparse autoencoders useful for java function bug detection?" 2025. [Online]. Available: https://arxiv.org/abs/2505.10375

[71] A. Zou, L. Phan, S. Chen, J. Campbell, P. Guo, R. Ren, A. Pan, X. Yin, M. Mazeika, A.-K. Dombrowski, S. Goel, N. Li, M. J. Byun, Z. Wang, A. Mallen, S. Basart, S. Koyejo, D. Song, M. Fredrikson, J. Z. Kolter, and D. Hendrycks, "Representation engineering: A top-down approach to ai transparency," 2025. [Online]. Available: https://arxiv.org/abs/2310.01405

[72] W. Yu, R. Mangal, T. Zhuo, M. Fredrikson, and C. S. Pasareanu, "A mixture of linear corrections generates secure code," *arXiv preprint arXiv:2507.09508*, 2025.

[73] G. Alain and Y. Bengio, "Understanding intermediate layers using linear classifier probes," *arXiv preprint arXiv:1610.01644*, 2016.