A Mixture of Linear Corrections Generates Secure Code

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) have become proficient at sophisticated code-generation tasks, yet remain ineffective at reliably detecting or avoiding code vulnerabilities. Does this deficiency stem from insufficient learning about code vulnerabilities, or is it merely a result of ineffective prompting? Using representation engineering techniques, we investigate whether LLMs internally encode the concepts necessary to identify code vulnerabilities. We find that current LLMs encode precise internal representations that distinguish vulnerable from secure code–achieving greater accuracy than standard prompting approaches. Leveraging these vulnerability-sensitive representations, we develop an inference-time steering technique that subtly modulates the model's token-generation probabilities through a mixture of corrections (MoC). Our method effectively guides LLMs to produce less vulnerable code without compromising functionality, demonstrating a practical approach to controlled vulnerability management in generated code. Notably, MoC enhances the security ratio of Qwen2.5-Coder-7B by 8.9%, while simultaneously improving functionality on HumanEval pass@1 by 2.1%.

1 Introduction

Large language models (LLMs) have rapidly become useful tools for developers, demonstrating remarkable proficiency across a wide array of code generation tasks [1, 2]. Current LLMs excel at understanding complex programming concepts [3], generating syntactically correct and functionally relevant code [4, 5], and even providing explanations, optimizations, and debugging assistance [6].

Despite these advances, even state-of-the-art models exhibit significant limitations with identifying vulnerable code. Our empirical analysis (Figure 1) on different sizes of CodeLlama [7] and Qwen2.5-Coder [4] reveals that traditional prompting techniques, including few-shot exemplars and detailed Common Weakness Enumeration (CWE) descriptions, result in accuracy comparable to random guessing (50%). Surprisingly, increasing the model parameter count fails to reliably improve detection accuracy, suggesting a persistent gaps between increased coding capabilities and the closely related task of identifying and generating *secure* code.

This motivates the question: *do code-generating LLMs inherently lack the knowledge to differentiate between vulnerable and secure code, or is this knowledge simply not accessible via prompting?* Using linear probing [8, 9], we find that LLMs do indeed possess latent representations that distinguish secure from vulnerable code far more effectively than standard prompts. Thus, despite these models' apparent lack of proficiency at the task of identifying vulnerable code, it is possible to access models' precise learned knowledge about vulnerabilities to accurately perform identification during inference, without the need for more expensive methods involving fine-tuning [10].

Building on this insight, we next investigate whether these latent representations can be leveraged during code generation. Specifically, we explore how to compute *correction vectors*–derived directly

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

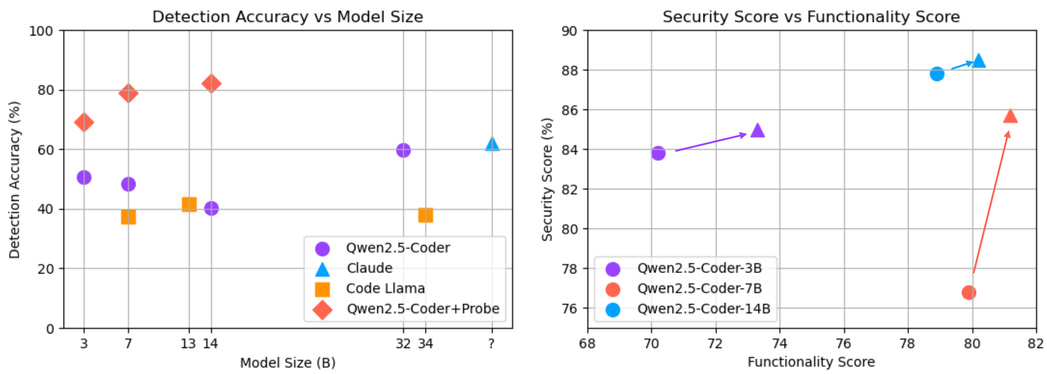

Figure 1: Left: The state-of-the-art code generation models cannot achieve high accuracy by purely prompting, while using probing can improve the accuracy. Right: Security and functional improvements by adding the mixture of corrections.

from clusters, linear probes, or through auxiliary neural networks–that encode vulnerability distinctions. These vectors, computed separately for individual CWEs, create a *mixture* of precise linear corrections.

We integrate these guiding vectors into the model's token generation process, applying conditional corrections with a temporal decay to subtly adjust next-token probabilities based on vulnerability, as assessed from the linear probes. This method enables granular, controlled steering of generation away from vulnerable code while avoiding interference with generation unlikely to yield vulnerable code, and thus without sacrificing functionality. Importantly, we show that it is also possible to apply this process *adversarially* to deliberately increase the likelihood of generating vulnerable code; this may be useful when training future models not to generate vulnerable code.

Our evaluation shows that this conditional steering not only significantly improves the security ratio (8.9% on Qwen2.5-Coder), but also frequently enhances the functional correctness of the resulting code (2.1% on HumanEval) (Figure 1). Moreover, we observe that the guiding vectors often *transfer* across models: vectores derived from one model can improve security in code generated by another model, such as the Qwen-2.5 variants. This transferability yields a computationally efficient way to harden models that are not well trained specifically on code data.

2 Related Work

LLM-assisted Vulnerability Detection Vulnerability detection is a crucial task in the field of computer security. Its primary objective is to identify potential software security threats, thus reducing the risk of cyber-attacks. LLMs have been explored for vulnerability detection in source code using two main paradigms: fine-tuning and prompt engineering [11]. Fine-tuning approaches typically introduce a binary classification head on top of the LLM and jointly optimize all model parameters using labeled vulnerable and secure code examples [12]. This setup has been applied across various Transformer architectures, including encoder-only [13, 14], encoder-decoder [15], and decoder-only models [16]. Some methods [17] also use Graph Neural Network backbone to extra features, and concatenate with LLM extracted features. Prompting-based methods [18] instead query powerful, often proprietary LLMs like GPT-4 using crafted natural language prompts. While these techniques have shown promising results on synthetic datasets [19], their performance on real-world vulnerability detection tasks is mixed. More recent work has explored structured prompting strategies, such as variations of Chain-of-Thought (CoT) prompting [20], and task-specific prompting frameworks targeting vulnerabilities like Use-Before-Initialization [21] and smart contract bugs [22]. Despite these advances, empirical studies have shown that both fine-tuning and prompting-based methods still struggle with vulnerability detection tasks [11].

In this work, we propose a new direction: instead of relying on extra finetuning or prompting strategies, we focus on representation engineering of trained LLMs to improve their internal understanding of secure and vulnerable code; we aim to enhance the latent representations used by the model to reason about code, enabling more robust vulnerability detection without introducing costly retraining or additional inference-time prompt engineering.

LLM-assisted Secure Code Generation Recent advancements in large language models (LLMs) have demonstrated significant potential in automating code generation tasks. However, the security of the code produced by these models remains a pressing concern, prompting a surge of research into security-aware techniques. Various approaches have been proposed to enhance the security of code generated by LLMs, focusing on both training-time and inference-time interventions.

Techniques such as SafeCoder [23] and ProSec [24] use security-centric fine-tuning to improve security, utility, and alignment. APILOT [25] addresses the challenge of outdated or insecure API usage by implementing a mechanism that navigates LLMs to generate secure, version-aware code, thereby reducing potential security threats associated with deprecated APIs. INDICT [26] presents a multi-agent framework that employs internal dialogues between safety-driven and helpfulness-driven critics to iteratively refine code generation, enhancing both the security and functionality of the output. CodeFavor [27] proposes a code preference model trained on synthetic evolution data, including code commits and critiques, to predict whether a code snippet adheres to secure coding practices. While it does not directly generate code, it provides a mechanism to evaluate and prefer secure code snippets. SVEN [28] is closest to our work; it introduces a method that guides LLMs to generate secure or insecure code by learning a continuous prompt, without modifying the model's weights. This approach allows for controlled code generation based on specified security properties. In contrast to SVEN, we compute a mixture of correction vectors, which are applied conditionally, leading to better control of the code generation. We compare in more detail with SVEN in section 4.

Steering & Controlling LLM Generation Controllable generation refers to the ability to steer the outputs of large language models (LLMs) toward desired properties, such as stylistic attributes, factuality, safety, or personalization. A growing body of work has focused on developing techniques for controlling LLMs both at the input and internal representation levels [29]. A prominent strategy for understanding and influencing LLM behavior is probing, which involves training lightweight classifiers on the model's internal activations to extract human-interpretable features [8]. Probing has been widely used to reveal latent knowledge in language models and, more recently, to guide and steer generation behavior by identifying representation subspaces associated with specific attributes. Recent advances in representation engineering go beyond passive probing, proposing direct interventions in the model's latent space. These techniques identify semantically meaningful directions in activation space and apply steering vectors to modify model behavior without full retraining [30, 9]. Such approaches have been used for tasks like factuality correction, sentiment control, and personalized generation [31, 32]. Despite this progress, only a few studies [28, 33] have explored the application of controllable generation techniques to code generation, where correctness, determinism, and alignment with developer intent are critical. In this paper, we propose a novel framework that applies probing and representation interventions to code generation models. Our method performs conditional interventions in the activation space, guided by the outputs of probes trained to detect semantic properties or vulnerabilities in the code.

3 A Mixture of Linear Corrections

Figure 2 gives an overview of our approach. We first train a set of linear probes for code vulnerability detection, as described in subsection 3.1, which we then use alongside one of four methods to obtain a set of linear corrections, as described in subsection 3.2. Finally, we use a mixture of the corrections, one for each vulnerability class, to generate secure code, as described in subsection 3.3.

3.1 Vulnerable Code Detection Using Linear Probing

We are given a decoder model used for code generation, denoted $\mathcal{G}$, with transfomer blocks $L_0, \ldots L_n$. We denote the d-dimensional activations of the hidden state at the last token position of a block L_i as s_i. A *probe* is a diagnostic tool that analyzes the information represented s_i, for a particular block i. Given a dataset $\mathcal{D}$ of paired (secure, vulnerable) code samples $\{(x^+, x^-)\}$ and a vulnerability type j, we will write $\mathcal{D}_j$ to refer to the subset of $\mathcal{D}$ consisting only of vulnerability type j.

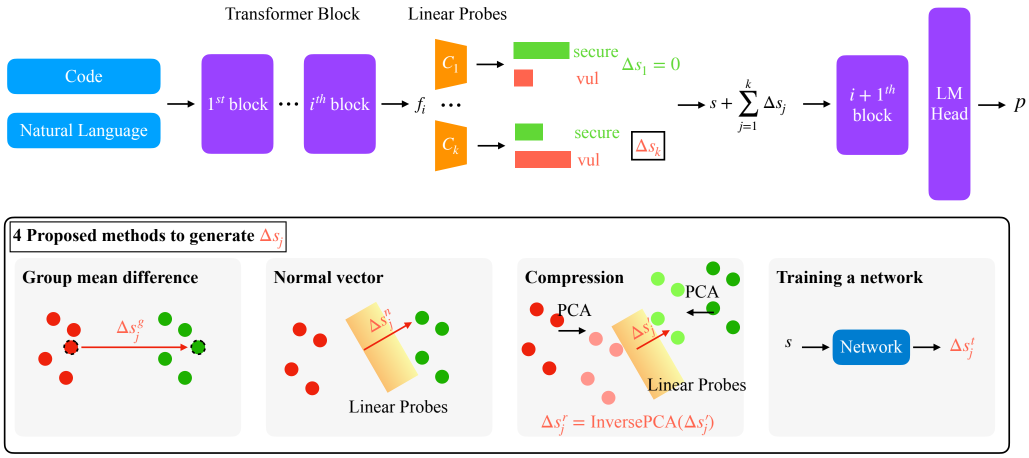

Figure 2: Mixture of corrections (MoC). There are four ways to obtain corrections for each vulnerability j. During inference, MoC applies correction $\Delta s_j^{c\in\{g,n,r,t\}}$ if the hidden states are at risk of generating j^{th} type of vulnerable code, and won't make correction if secure.

From the dataset, we use cross entropy loss to train a set of linear probes $c_0(\cdot), \cdots c_k(\cdot)$ against a binary label v which denotes whether the features s were produced by a vulnerable (x^-) or secure (x^+) sample (Equation 1).

$$\mathcal{L}_c = \mathrm{CE}(c(s), v) = \mathrm{CE}(Ws + b, v) \tag{1}$$

We perform this training on all blocks in the model, and identify the block L^* with the smallest loss to take as the final probe for each vulnerability.

Our empirical investigation reveals that the activations within an LLM exhibit remarkable efficacy for vulnerability detection and the detection accuracy is good compared to previous finetuned or GNN-based detectors. This finding is also a proof that hidden states within LLMs encode richer information than the terminal outputs, as shown in [34, 35], Notably, training these probes only requires minimal data with lightweight parameters.

3.2 Controlling Vulnerability Generation with a Mixture of Corrections (MoC)

The efficacy of vulnerability detection through representational probing demonstrates that the latent activations within transformer attention mechanisms encode substantive information pertaining to code security vulnerabilities. This observation suggests that these representations can be leveraged to guide code generation—-either to make code more secure, or to deliberately produce vulnerabilities. We propose a framework called *Mixture of Corrections* (MoC) for accomplishing this, which computes a set (mixture) of correction vectors Δs_j for each vulnerability class, which are subsequently combined with hidden states when generating code during inference.

We present four methods for computing correction vectors, both static, wherein Δs_j is a function only of the vulnerability type j, as well as dynamic, where it is also conditioned on the decoder's current hidden states. MoC is illustrated in Figure 2, and detailed in alg. 1.

3.2.1 Static Correction Vectors

Difference of group mean. The most direct and efficient way of measuring the correction is by computing the arithmetic differential between the centroid vectors of the respective class data samples, as shown in Equation 2.

$$\Delta s_j^g = \frac{1}{|s_j^+|}\sum_{\mathcal{D}_j} s_j^+ - \frac{1}{|s_j^-|}\sum_{\mathcal{D}_j} s_j^- \tag{2}$$

In Equation 2, $|\cdot|$ denotes the cardinality of the set of j^{th} vulnerability.

Normal vector of the decision boundary. The linear probe established in the preceding section effectively partitions the feature space into two disjoint subspaces via a linear hyperplane that constitutes the decision boundary. Thus, another way of computing the requisite correction is to traverse orthogonally from the vulnerable class subspace to the non-vulnerable class subspace—specifically, the normal vector to the decision boundary hyperplane. The decision boundary is $(W_{1,:} - W_{0,:})x + b_1 - b_0 = 0$, and the normal vector is characterized in Equation 3.

$$\Delta s_j^n = W_{1,:} - W_{0,:} \tag{3}$$

In Equation 3, $W \in R^{2\times d}$, and $W_{0,:}$ and $W_{1,:}$ is the first and second row of W.

Reduced normal vector. Direct utilization of LLM hidden states can exhibit susceptibility to overfitting phenomena and manifest training instability in probe training. Our assumption is that within the high-dimensional feature space, these features encode not only vulnerability-related information but also other types of information that can be considered as noise in code generation contexts. To mitigate these adverse effects, we use dimensionality reduction techniques, specifically principal component analysis (PCA), to derive a more robust correction vector [9]. Let s'_j to denote the compressed version of s_j, and we train a linear probe $c'_j(x) = W'x+b'$ on the compressed vectors, then project the correction back to the original high-dimensional space, i.e. $\Delta s_j^r = \text{PCAInverse}(c'_j)$, where $W' \in R^{2\times d'}$, d' is the reduced dimension.

3.2.2 Dynamic Correction Vectors

To condition the correction vectors additionally on the dynamic state of the model, for each vulnerability type we train a neural network $N(\cdot)$ that directly predicts the correction vector Δs^t. To train this network—given that the vulnerability dataset usually contains not only the paired data, but also the detailed line changes of the vulnerable lines of code—we add multiple aspects of supervision. Let p_j denote the $\mathcal{G}$'s output probability corresponding to s_j at the hidden space, and let y^+ denote the secure code labels in token space. Note that the p_j, s_j and y^+ here are per-token supervision, and should be denoted as $p_{j,m}, p_{j,m+1}, \cdots, p_{j,m+n}$, where m is the index for the start of the vulnerable code token, and $m+n$ is the index for the end of the vulnerable code token. We use the p_j for simplicity.

The training involves three loss terms, including mean square error, $\mathcal{L}_{mse} = \text{MSE}(s_j^-, s_j^+)$, cross entropy loss $\mathcal{L}_{ce} = \text{CE}(p_j^-, y_j^+)$, and KL-divergence, $\mathcal{L}_{KL} = \text{KL}(p_j^-, p_j^+)$. The final loss form is a combination of these supervisions, $\mathcal{L} = \beta_1\mathcal{L}_{mse} + \beta_2\mathcal{L}_{ce} + \beta_3\mathcal{L}_{KL}$. The network then gives the correction, as $\Delta s_j^t = N(s_j)$.

3.3 Inference with corrections

After obtaining the mixture of corrections $\{\Delta s_j\}$, one for each vulnerability class, we apply the corrections during inference time, by adding the linear combination of corrections to the hidden states when generating every token. However, unlike the works [36, 37] that directly add the vectors, i.e., $f = f + \Delta s$, we find it sub-optimal and add the following tricks.

Decay. During inference, as the generation becomes longer, the correction accumulates, which results in too much correction and the output generations can be less meaningful. To avoid such a large change in hidden states, we use a decay factor to gradually reduce the impact of the newly added correction during inference.

$$\Delta s := \alpha(t) \cdot \Delta s, \tag{4}$$

where t is the number of tokens newly generated during inference, e.g. when generating the first token, $t = 0$, and when generating the k^{th} token, $t = k - 1$. $\alpha(\cdot)$ is a negative exponential function.

Conditional correction. When generating secure code, there's no need to modify the hidden states and add the correction vectors. The corrections are only applied when the hidden states are at risk of generating vulnerable codes. Thus, we apply a conditional correction, as in the Figure 2, we first use the previously obtained linear probes to detect if the current hidden states are at risk of generating vulnerable codes of vulnerable type j, and then apply the corresponding correction Δs_j only if the hidden states are vulnerable. If the current hidden states shows multiple different vulnerabilities, then the corrections are added as a linear combination as shown in Equation 5.

$$s+= \begin{cases} \Delta s_j, & \text{if argmax}(c_j(s)) = 0 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Algorithm 1: Mixture of Corrections

Input: (1) A code generation LLM $\mathcal{G}$, and its i^{th} transformer blocks L_i; (2) A dataset of paired vulnerable and secure data $\mathcal{D} = \{\mathcal{D}_j\}$.
Output: A secure code generation x

`// Training stage`
foreach $j \in \{0, \ldots, k\}$ **do**
    Train the linear probe c_j according to subsection 3.1.
    Obtain the correction vector $\Delta s_j^{c \in \{g,n,r,t\}}$ according to subsubsection 3.2.1 & subsubsection 3.2.2
end
`// Inference stage`
foreach *token* x_{t+1} **do**
    $s = L^*(x_{1:t})$; `// Get the hidden states`
    foreach $j \in \{0, \ldots, k\}$ **do**
        if $(argmax(c_j(s)) = 0)$ **then**
            $s := s + \alpha(t) \cdot \Delta s_j^m$; `// add correction if vulnerable`
    end
end
return x

We present the overall MoC algorithm in alg. 1. MoC first trains the light-weight linear probes, and then obtains corrections for each type of vulnerability. During inference, MoC applies these corrections if the hidden states are at risk of generating vulnerable code, as measured by the linear probes.

4 Experiments

In this section, we first present the evaluation of the trained probe's efficacy in identifying vulnerable code, then we investigate whether the mixture of corrections improves, or adversarially, decreases security, in code generation, and the transferability of the corrections across models.

4.1 Vulnerable Code Detection

Dataset. Following SVEN [28], which contributes a high-quality pairwise code dataset of 9 different CWEs, we use this dataset as our training and evaluation set. In each vulnerability class, we random sample a train set and a subset. Due to the imbalance of the dataset across different types of vulnerabilities, we keep the evaluation set the same size, and the train set might be of different sizes. Notably, the vulnerable and secure data are balanced in our settings.

Evaluation Metric. Accuracy Acc_v (%) and Acc_s (%) is the accuracy of the vulnerable code and secure code respectively. For training, Acc (%) and F1 (range from 0 to 1) are the accuracy and F1-score on the evaluation set.

Linear Probe Details. For each vulnerability, the probe is trained on around 50 to 150 data points due to the imbalance of different types of vulnerabilities. The training epoch is from 50 to 200, with a batch size of 64, learning rate $5e-4$, SGD optimizer with a momentum and weight decay.

Training Details. One of the proposed correction methods requires training of another network, and the network structure is a three-layer multi-layer perception (MLP), with GeLU activation, layer normalization, and dropout layer. The learning rate is $1e-3$, with an Adam optimizer. To construct the training pair f_i^+ and f_i^-, we also use the 'line changes' information in each pair of the vulnerable-secure data for detailed supervision. We save the hidden states tensors before training so that training is done on single GPU even for 14B models.

RQ1: Can LLMs detect vulnerable code by direct prompting? As evidenced in Figure 1, by directly prompting the code generation LLMs, the accuracy is suboptimal. Notably, the prompt includes both few-shot examples sampled from the same dataset (two positive examples and two negative examples), and a description of the specific vulnerability (e.g. CWE-022). We list pervulnerability experimental results in subsection A.1. We can draw the following conclusions: (1) In general, the current code-related LLMs, including Qwen2.5-Coder series and CodeLlama series, and closed-source model Claude lack the ability to detect code vulnerabilities by prompting. Possible reasons are that the vulnerabilities are less focused on and that these models are not specifically trained on vulnerability code data. (2) There is no clear relation between the model size and their vulnerable detection capacity, though the 32B or 34B models show a small performance improvement compared to smaller models. (3) QC-7B, 14B and 32B, CL-34B models tend to predict the code secure. For the QC-7B, 14B, and CL series, the accuracy is no better than a random guess.

Table 1: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid means the output of the LLM doesn't follow the format or the output does not include any answers. QC is in short for Qwen2.5-Coder, and CL for Code Llama.

	Acc_v	Acc_s	Invalid
QC-3B	51	51	0
QC-7B	23	74	3
QC-14B	25	55	27
QC-32B	30	81	0
CL-7B	64	10	29
CL-13B	44	43	14
CL-34B	21	59	27
Claude	63	43	0

Prompts

// CWE description
CWE-022, commonly called "Path Traversal," is a vulnerability when an application fails to appropriately limit . . .
// Few-shot examples
For example, 'code1 . . . ' is vulnerable, while 'code2 . . . ' is not vulnerable. 'code3 . . . ' is vulnerable, while 'code4 . . . ' is not vulnerable.
// Prompt
Is the subsequent code susceptible to the specified vulnerability?
// Test code
code . . .
Answer the question with simply yes or no.

Table 2: Performance on code vulnerability detection.

Method	QC-3B		QC-7B		QC-14B	
	Acc	F1	Acc	F1	Acc	F1
Prompting	51	0.56	49	0.53	40	0.37
Linear Probe W/O Few-shot	66	0.65	68	0.66	75	0.79
Linear Probe	69	0.63	**79**	**0.76**	**82**	**0.85**
Linear Probe PCA	**72**	**0.68**	76	0.74	78	0.80
MLP Probe	**72**	0.66	77	0.75	80	0.80

RQ2: Can hidden states within LLMs help detect vulnerable code? In Table 2, 'Prompting' means no probe training, and just prompting by few-shot and descriptions as in RQ1. 'Linear Probe W/O Few-shot' refers to, when getting hidden states f from L_i in $\mathcal{G}$, the input only includes the code without few-shot examples. The other probes' input all includes few-shot examples. 'Linear Probe' and 'Linear Probe PCA' contains a linear layer with a weight matrix W and bias b, the difference is without PCA $W \in R^{2\times d}$, where $d = 3168$ in this cases, with PCA, $W' \in R^{2\times d'}$ and d' is a number between 50 to 100. 'MLP probe' contains 2 or 3 multi-layer perceptron layers, each layer includes a linear layer, a ReLU activation function, a layer norm, and a dropout layer.

From Table 2, we can draw the following conclusion. (1) Overall, probing methods can detect vulnerable code, showing that hidden states within LLMs actually contain vulnerability-related information. (2) Using few-shot examples in the text prompt improves the vulnerability detection, showing that the prompting techniques help with the hidden states probing. (3) MLP probes, even with more parameters, don't show a clear improvement compared to linear probes. This may be due to the simplicity of the task: it is a classification task and linear probes are enough to distinguish the secure and vulnerable classes. (4) The performance shows a relation with the LLM scale, as the LLM becomes larger, the performance of the probe gets higher.

Table 3: Performance on code generation. SR_h ($\uparrow$)(%) and SR_w ($\downarrow$)(%) denote the security ratio when applying hardening and weakening. HE denotes HumanEval pass@1.

	QC-3B			QC-7B			QC-14B		
	SR_h	SR_w	HE	SR_h	SR_w	HE	SR_h	SR_w	HE
Base Model	83.8	83.8	70.2	76.8	76.8	79.9	87.8	87.8	78.9
SVEN	-	-	-	65.0	54.0	75.3	69.7	65.4	75.0
Group Mean Diff	84.7	78.8	73.9	84.0	75.5	81.4	88.5	87.1	80.2
Normal Vector	83.3	81.0	74.5	84.3	80.4	82.0	87.5	87.2	78.9
Normal Vector PCA	85.0	82.4	73.3	82.9	78.9	82.0	88.3	82.5	78.3
Dynamic NN-based	84.9	82.1	70.8	85.7	75.9	81.2	88.0	87.1	82.0

4.2 Secure Code Generation

Evaluation. We evaluate the code security using GitHub CodeQL [38], which is an open-source code security analyzer that can detect different vulnerabilities based on the custom queries. We report the security rate SR (%). SR_h means hardening the security (the higher the better), while SR_w means weakening the security (the lower the better). The generated code is considered secure only if it doesn't contains any main CWEs based on CodeQL. Note that we test the proposed methods on the SVEN test set, which is different from the evaluation set in subsection 4.1. For code functionality, we test the pass@1 on HumanEval.

RQ3: Can the mixture of corrections help in secure code generation? In Table 3, 'Base Model' means applying no corrections. 'SVEN' [28] is a method that trains prefix soft embeddings and concatenates the embeddings to the LLM during inference. However, on the 3B model, the training loss doesn't decrease, so we choose not to report the results. Then the four correction methods refer to $\Delta s_j^g, \Delta s_j^n, \Delta s_j^r, \Delta s_j^t$ respectively. In the security hardening cases, the conditional generation is utilized, while in the security weakening cases, since the aim is to modify the secure hidden states to insecure ones, and thus it is not conditional, we add the sum of the negative corrections to it, i.e. $\Delta s = \sum_{j=1}^{k} -\alpha(t)s_j$.

We can draw the conclusion that: (1) Generally, applying MoC can improve not only the security but also the functionality of the code. (2) On Qwen-2.5-Coder-7B, the dynamic NN-based method outperforms others. (3) There are some cases when the weakening cases do not actually bring out more vulnerable codes. One possible guess is that, since we use the sum of all the correction vectors, they may suffer a bit by canceling out on some critical directions. (4) In most cases, the functionality of the LLMs is not affected and even shows some improvements.

RQ4: Probes at which attention blocks are the best? We train the probe on different attention blocks, and test their effect on the code generation. As in Figure 3, the last attention block shows the best performance.

Ablation Study. We conduct two versions for PCA corrections. One version is to first obtain both the decision boundary and compute the normal vector to the decision boundary in the compressed space, and then project the normal vector back to the high-dimensional space, as follows:

$$\Delta s_j^r = \text{PCAInverse}(W'_{1,:} - W'_{0,:}) = (W'_{1,:} - W'_{0,:})V + M, \tag{6}$$

where V are the principal components and M are the mean of the vectors. Another is to first project the weighting matrix back and then calculate the normal vector, as follows:

$$\Delta s_j^r = \text{PCAInverse}(W')_{1,:} - \text{PCAInverse}(W')_{0,:} = (W'V)_{1,:} - (W'V)_{0,:} + M_{1,:} - M_{0,:}, \tag{7}$$

note that $W \neq W'V$. We tried both, as in Table 4, the first PCA version fails to generate reasonable outputs, while the second PCA implementation can bring improvements, suggesting that the hidden states space within LLMs is elaborate.

Table 4: Ablation study on how to obtain PCA correction.

	SR_h	SR_w	HE
Base Model	76.8	76.8	79.9
Δs_j^r in Equation 6	6.3	4.6	19.8
Δs_j^r in Equation 7	82.9	78.9	82.0

Table 5: Ablation study on conditional correction and decay.

	SR_h	HE
Base Model	76.8	79.9
Normal Vector W/O Condition	81.7	77.0
Normal Vector W/O Decay	85.8	69.6
Normal Vector	84.3	82.0

Ablation in Table 5 is conducted on QC-7B model. (1) Adding conditions improves both the secure ratio and the functionality. (2) Though adding decay results in an improvement on secure ratio, it affects the functionality significantly.

RQ5: Can the corrections learned for one model transfer to another? We try to apply the corrections learned from one model and apply them on another model. As in Table 6, the corrections are trained on Qwen2.5-Coder model and implemented on the Qwen2.5-Instruct model, where they share the same hidden dimension and the same model structure. We find that it shows some level of transferability in 3B and 7B models, but not on larger model. However, the functionality of the model based on transferred corrections are harmed on larger models.

Table 6: Transferability across models. We use QI in short for Qwen2.5-Instruct and QC for Qwen2.5-Coder. HE is short for HumanEval. The corrections are Normal Vector obtained from QC models.

	Corrections	SR_h	SR_w	HE
QI-7B		76.8	76.8	69.6
QI-7B	QC-7B	77.1	76.3	65.8
QI-3B		75.3	75.3	54.0
QI-3B	QC-3B	78.0	71.2	54.7
QI-14B		63.6	63.6	74.5
QI-14B	QC-14B	58.2	53.5	72.7

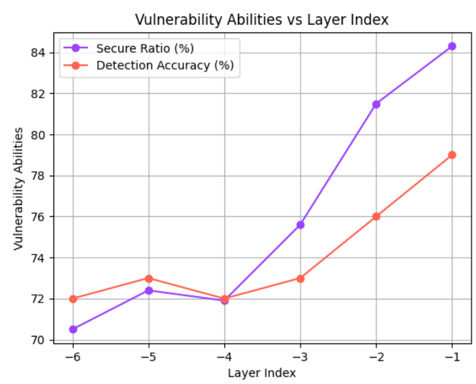

Figure 3: Ablations on i^{th} attention blocks.

RQ6: Why can the MoC gain the improvement on functionality for free? As in Table 3, we observe a consistent improvement on the functionality score except for the 14B model. A possible reason for this is that the more buggy codes have a higher possibility of also being vulnerable code [39], and there are overlaps between bug-prone code and vulnerabilities [40].

5 Conclusion

Our investigation reveals that code generation LLMs encode vulnerability-discriminative information in their hidden representations, accessible through lightweight linear probes. We leveraged this insight to develop a Mixture of Linear Corrections (MoC) framework that conditionally applies guiding vectors during inference to enhance code security. Experimental results show our method effectively improves both security ratios and functional correctness across multiple model sizes and demonstrates transferability between models. This work provides a computationally efficient approach to secure code generation without requiring costly retraining or extensive prompt engineering, opening new avenues for representation-based security interventions in generative AI.

Limitations. Currently we use every probe in every token generation, which consumes more time than the base model. Further work can implement all probes in parallel to accelerate. Moreover, the MoC can only address known code vulnerabilities, further research can focus on using hidden states to explore undiscovered vulnerabiliies.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[2] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[3] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

[4] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[5] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025.

[6] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022.

[7] Wenhan Xiong Grattafiori, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[8] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. 2017.

[9] Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *CoRR*, 2023.

[10] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

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

[12] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *ACL (Findings)*, 2024.

[13] Soolin Kim, Jusop Choi, Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. Vuldebert: A vulnerability detection system using bert. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 69–74. IEEE, 2022.

[14] Xiaobing Sun, Liangqiong Tu, Jiale Zhang, Jie Cai, Bin Li, and Yu Wang. Assbert: Active and semi-supervised bert for smart contract vulnerability detection. *Journal of Information Security and Applications*, 73:103423, 2023.

[15] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 935–947, 2022.

[16] Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51, 2024.

[17] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

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

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

[20] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 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, 2024.

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

[22] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[23] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. In *Forty-first International Conference on Machine Learning*, 2024.

[24] Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. Prosec: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*, 2024.

[25] Weiheng Bai, Keyang Xuan, Pengxiang Huang, Qiushi Wu, Jianing Wen, Jingjing Wu, and Kangjie Lu. Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. *arXiv preprint arXiv:2409.16526*, 2024.

[26] Hung Le, Doyen Sahoo, Yingbo Zhou, Caiming Xiong, and Silvio Savarese. Indict: Code generation with internal dialogues of critiques for both security and helpfulness. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[27] Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024.

[28] Jingxuan He and Martin Vechev. 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, 2023.

[29] Xun Liang, Hanyu Wang, Yezhaohui Wang, Shichao Song, Jiawei Yang, Simin Niu, Jie Hu, Dan Liu, Shunyu Yao, Feiyu Xiong, and Zhiyu Li. Controllable text generation for large language models: A survey, 2024.

[30] Jan Wehner, Sahar Abdelnabi, Daniel Tan, David Krueger, and Mario Fritz. Taxonomy, opportunities, and challenges of representation engineering for large language models. *arXiv preprint arXiv:2502.19649*, 2025.

[31] Yuanpu Cao, Tianrong Zhang, Bochuan Cao, Ziyi Yin, Lu Lin, Fenglong Ma, and Jinghui Chen. Personalized steering of large language models: Versatile steering vectors through bi-directional preference optimization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[32] Tianlong Wang, Xianfeng Jiao, Yinghao Zhu, Zhongzhi Chen, Yifan He, Xu Chu, Junyi Gao, Yasha Wang, and Liantao Ma. Adaptive activation steering: A tuning-free llm truthfulness improvement method for diverse hallucinations categories. In *Proceedings of the ACM on Web Conference 2025*, pages 2562–2578, 2025.

[33] Sameer Pimparkhede, Mehant Kammakomati, Srikanth Tamilselvam, Prince Kumar, Ashok Kumar, and Pushpak Bhattacharyya. Doccgen: Document-based controlled code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18681–18697, 2024.

[34] Amos Azaria and Tom Mitchell. The internal state of an llm knows when it's lying. *arXiv preprint arXiv:2304.13734*, 2023.

[35] Junhao Chen, Shengding Hu, Zhiyuan Liu, and Maosong Sun. States hidden in hidden states: Llms emerge discrete state representations implicitly. *arXiv preprint arXiv:2407.11421*, 2024.

[36] Amrita Bhattacharjee, Shaona Ghosh, Traian Rebedea, and Christopher Parisien. Towards inference-time category-wise safety steering for large language models. In *Neurips Safe Generative AI Workshop 2024*, 2024.

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

[38] CodeQL. Github codeql, 2025. `https://github.com/github/codeql`.

[39] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015.

[40] Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 269–279. IEEE, 2015.

A Appendix

A.1 Detailed experimental results on direct prompting for vulnerability detection.

Here we show the detailed experimental results for each CWEs for each model. The results are from the Table 7 to Table 13, we find that:

1. Difference CWE types shows every different trends, for example, the CWE-125, CWE-190, CWE-416, CWE-476, CWE-787 contains mostly codes in language c, and Qwen-25-Coder series tend to think they are safe, as in Table 8, Table 9 and Table 10. And the CodeLlama series tend to regard the CWE-416, CWE-476 and CWE-787 as safe, as in Table 12 and Table 13.
2. Overall, QwenCoder series are more recently developed and shows better abilities than CodeLlama series. And overall, the QC series show a better instruction following ability than CL series, as the Invalid rates are lower.

Table 7: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid$_v$ and Invalid$_s$ mean the output of the LLM doesn't follow the format or the output does not include any answers.

QwenCoder-3B				
Vul-type	Acc$_v$	Acc$_s$	Invalid$_v$	Invalid$_s$
22	49	49	0	0
78	51	43	1	0
79	35	58	0	0
89	47	61	0	1
125	52	45	1	0
190	67	53	0	3
416	45	62	0	0
476	59	44	1	1
787	50	42	0	0
Ave	51	51	0	1

Table 8: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid$_v$ and Invalid$_s$ mean the output of the LLM doesn't follow the format or the output does not include any answers.

QwenCoder-7B				
Vul-type	Acc$_v$	Acc$_s$	Invalid$_v$	Invalid$_s$
22	43	53	0	0
78	66	72	0	0
79	44	42	5	5
89	1	69	0	1
125	12	77	13	9
190	8	100	0	0
416	2	96	0	0
476	13	84	4	3
787	15	73	3	2
Ave	23	74	3	2

A.2 Detailed experimental results on probing for vulnerability detection.

Here we shows more results about detailed per-CWE results on detection when training a linear probe on the last attention block. We can draw the conclusion:

1. Overall, the non-PCA probe in Table 14 shows better results than PCA reduced probes in Table 15. Possible reasons are that the PCA reduced too much information that may be essential for vulnerability detection.

Table 9: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid_v and Invalid_s mean the output of the LLM doesn't follow the format or the output does not include any answers.

QwenCoder-14B				
Vul-type	Acc_v	Acc_s	Invalid_v	Invalid_s
22	2	31	69	69
78	25	62	12	20
79	28	72	14	14
89	13	8	78	91
125	31	64	13	14
190	36	47	19	11
416	27	71	16	15
476	32	74	3	1
787	35	63	23	6
Ave	25	55	27	27

Table 10: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid_v and Invalid_s mean the output of the LLM doesn't follow the format or the output does not include any answers.

QwenCoder-32B				
Vul-type	Acc_v	Acc_s	Invalid_v	Invalid_s
22	43	45	0	0
78	69	71	0	0
79	56	44	2	2
89	99	68	0	1
125	2	100	0	0
190	0	100	0	0
416	0	100	0	0
476	0	99	0	1
787	2	100	0	0
Ave	30	81	0	0

2. Overall, the CWE-022, CWE-078, CWE-079, and CWE-089 are mostly based on python language. And these shows a higher accuracy than other CWEs, especially on QC-14B and 7B models.
3. The CWE-125 and CWE-476 are kind of hard to detect, especially, as the model gets larger, the accuracy on these two CWE-types are not getting higher, which indicates that their vulnerable features are harder to extract.

A.3 Boarder Impact

Our work on code vulnerability detection and correction has significant societal implications. **Positively**, it enhances code security, potentially reducing data breaches and cyberattacks that impact millions of users annually. Secure code generation tools democratize cybersecurity expertise, benefiting resource-constrained organizations and critical infrastructure. **Negatively**, adversarial applications of our techniques could be used to deliberately introduce subtle vulnerabilities or automate exploitation of existing weaknesses. Additionally, over-reliance on automated security tools may create false confidence and reduce human oversight. We encourage responsible deployment with human-in-the-loop verification and recommend against using these methods in high-risk applications without thorough testing.

Table 11: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid$_v$ and Invalid$_s$ mean the output of the LLM doesn't follow the format or the output does not include any answers.

CodeLlama-7B				
Vul-type	Acc$_v$	Acc$_s$	Invalid$_v$	Invalid$_s$
22	63	0	41	37
78	49	8	48	48
79	44	9	49	51
89	0	0	100	100
125	89	11	2	2
190	86	19	3	3
416	89	9	4	2
476	59	32	12	12
787	100	2	0	0
Ave	64	10	29	28

Table 12: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid$_v$ and Invalid$_s$ mean the output of the LLM doesn't follow the format or the output does not include any answers.

CodeLlama-13B				
Vul-type	Acc$_v$	Acc$_s$	Invalid$_v$	Invalid$_s$
22	86	14	4	4
78	63	29	10	10
79	47	42	16	21
89	65	13	26	21
125	20	18	63	62
190	83	19	0	0
416	5	96	2	0
476	8	72	2	3
787	20	89	0	1
Ave	44	43	14	13

Table 13: Accuracy of vulnerable code detection by direct prompting the LLM. Invalid$_v$ and Invalid$_s$ mean the output of the LLM doesn't follow the format or the output does not include any answers.

CodeLlama-34B				
Vul-type	Acc$_v$	Acc$_s$	Invalid$_v$	Invalid$_s$
22	61	49	8	6
78	55	37	17	15
79	7	2	90	90
89	53	60	2	3
125	1	54	49	46
190	0	89	19	17
416	2	85	15	15
476	0	81	21	22
787	10	75	23	19
Ave	21	59	27	26

Table 14: Performance on code vulnerability detection.

	QC-3B		QC-7B		QC-14B	
Vul-type	Acc	F1	Acc	F1	Acc	F1
22	0.8	0.83	0.9	0.89	0.7	0.73
78	0.7	0.67	0.9	0.91	0.9	0.91
79	0.7	0.67	0.9	0.89	0.9	0.89
89	0.8	0.75	0.8	0.75	1	1
125	0.7	0.73	0.7	0.67	0.6	0.67
190	0.7	0.57	0.6	0.67	0.8	0.8
416	0.8	0.75	0.7	0.57	0.8	0.8
476	0.6	0.6	0.6	0.67	0.6	0.67
787	0.7	0.57	0.7	0.67	0.7	0.73
Ave	0.72	0.68	0.76	0.74	0.78	0.8

Table 15: Performance on code vulnerability detection. PCA is applied to the hidden states.

	QC-3B		QC-7B		QC-14B	
Vul-type	Acc	F1	Acc	F1	Acc	F1
22	0.6	0.33	0.8	0.75	0.8	0.8
78	0.7	0.57	0.9	0.89	0.9	0.91
79	0.6	0.5	0.9	0.91	1	1
89	0.8	0.75	0.9	0.89	1	1
125	0.6	0.67	0.6	0.33	0.6	0.67
190	0.7	0.57	0.8	0.83	0.9	0.89
416	0.8	0.75	0.7	0.77	0.8	0.8
476	0.8	0.83	0.7	0.67	0.6	0.71
787	0.6	0.71	0.8	0.8	0.8	0.83
Ave	0.69	0.63	0.79	0.76	0.82	0.85

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Yes, as in the abstract and introduction.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: As in the section Conclusion.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.

- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: As in the Probe details, training details.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The dataset in a public dataset, while the code will be public later.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.

- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: As in the experiments section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: In the experiment section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: In experiment section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.

- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss this in the appendix.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [No]

    Justification: It can be used adversarially.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Yes

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [NA]

Justification: Does not release new assets.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: Does not use crowdsourcing.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: We use LLMs for writing, and for prompting to get the results as stated in experiments.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.