Beyond Natural Language Perplexity: Detecting Dead Code Poisoning in Code Generation Datasets

Anonymous ACL submission

Abstract

The increasing adoption of large language models (LLMs) for code-related tasks has raised concerns about the security of their training datasets. One critical threat is *dead code poisoning*, where syntactically valid but functionally redundant code is injected into training data to manipulate model behavior. Such attacks can degrade the performance of neural code search systems, leading to biased or insecure code suggestions. Existing detection methods, such as token-level perplexity analysis, fail to effectively identify dead code due to the structural and contextual characteristics of programming languages. In this paper, we propose DEPA (Dead Code Perplexity Analysis), a novel line-level detection and cleansing method tailored to the structural properties of code. DEPA computes *line-level perplexity* by leveraging the contextual relationships between code lines and identifies anomalous lines by comparing their perplexity to the overall distribution within the file. Our experiments on benchmark datasets demonstrate that DEPA significantly outperforms existing methods, achieving up to a 160% improvement in detection accuracy and a six-fold increase in poisoned segment localization precision. Furthermore, DEPA enhances detection speed by 60%, making it practical for large-scale dataset cleansing. Overall, by addressing the unique challenges of dead code poisoning, DEPA provides a robust and efficient solution for safeguarding the integrity of code generation model training datasets.

1 Introduction

Large language models (LLMs) specialized for coding, often called Code LLMs (Lu et al., 2021; Roziere et al., 2023; Team et al., 2024), are extensively used for tasks such as code summarization (Ahmed and Devanbu, 2022), code completion (Zhang et al., 2024), and code search (Chen et al., 2024). As these models become more integrated into diverse development processes, protecting their training data becomes increasingly critical.

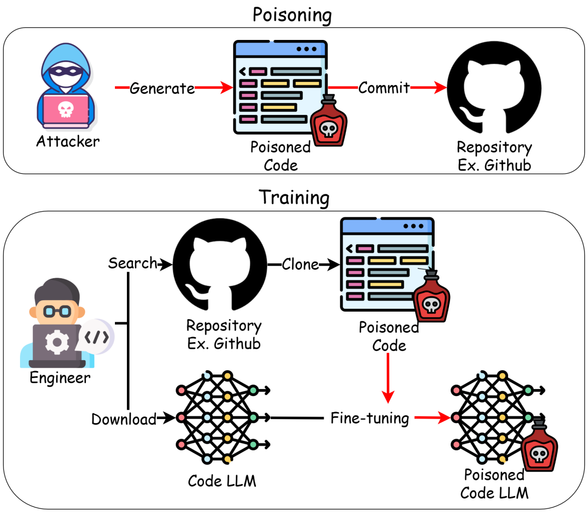

Figure 1: Data poisoning attack scenario.

In this context, data poisoning attacks commonly involve injecting *dead code* (Ramakrishnan and Albarghouthi, 2022; Wan et al., 2022), which consists of syntactically valid yet non-functional code snippets that act as triggers to alter model outputs. Such *dead code poisoning* can produce flawed, inefficient, or even malicious code suggestions, thereby undermining code search. Wan et al. (2022) demonstrated that selecting frequently used keywords in vulnerable code and pairing them with dead code can bias the model toward favoring insecure or defective code. Figure 1 illustrates how poisoned samples ultimately lead to a compromised Code LLM.

Detecting and removing dead code is challenging. In natural language, methods like ONION (Qi et al., 2021) rely on GPT-2 perplexity scores (Radford et al., 2019) to identify abnormal tokens indicating backdoor triggers. However, standard *word-level perplexity* methods designed for natural language do not directly apply to code. Although

Yang et al. (2024) tested ONION for detecting poisoned code (Yang et al., 2024; Ramakrishnan and Albarghouthi, 2022), its low detection accuracy at the code level made it ineffective for identifying dead code.

In studying dead code poisoning, we observed three key points. First, code has a structural rigidity absent in natural language; each line typically represents a discrete operational unit. Thus, anomalies from dead code are more evident at the line level than at the token level. Second, dead code does not affect program execution, making it functionally redundant yet strategically used as a backdoor trigger. Its impact is therefore more apparent when analyzing entire lines rather than individual tokens. Third, focusing on a single line's perplexity in isolation can be misleading, since a line may appear anomalous alone but be valid within the broader context. Hence, comparing each line's perplexity to the file's overall distribution is crucial to distinguish real anomalies from benign variations.

Guided by these insights, we first introduce a line-level perplexity measure tailored for code. We then propose **De**ad code **P**erplexity **A**nalysis (DEPA), a new detection and cleansing method designed around the structural properties of code. Unlike traditional word-level perplexity approaches, DEPA evaluates each line as a functional unit and compares its line-level perplexity against the overall file distribution, making it more effective at revealing dead code triggers that might otherwise remain hidden. Our experimental results show that DEPA substantially outperforms token-level approaches across multiple metrics. For example, DEPA achieves an F1-score of 0.26, compared to 0.10 for ONION(CodeGPT) and 0.17 for ONION(CodeLlama). In terms of precision for locating dead code within poisoned segments, DEPA reaches 0.87, whereas ONION(CodeGPT) and ONION(CodeLlama) achieve 0.33 and 0.20, respectively.

Overall, our contributions are as follows:

- We introduce DEPA, a line-level detection method guided by the structural characteristics of code. By incorporating contextual information into line-level perplexity calculations, DEPA improves anomaly detection without disrupting the overall code structure.

- Compared to ONION, DEPA improves the detection accuracy of poisoned code fragments by 0.72, raises the AUROC by 0.18, and increases detection speed by 60%.

2 Related Work

Data Poisoning on Code LLMs With the growing adoption of Code LLMs, concerns about training data security have intensified. For instance, OWASP has labeled *Data and Model Poisoning* as a critical threat.[1] Various studies highlight different attack vectors in Code LLMs. Sun et al. (2023); Yang et al. (2024) implant backdoors by substituting variable or method names with specific triggers, while others (Wan et al., 2022; Ramakrishnan and Albarghouthi, 2022) insert dead code into training data.

Poisoning Defense on Code LLMs Several defense mechanisms have been introduced to combat data poisoning in code. One widely used technique is spectral signature analysis (Tran et al., 2018), which detects anomalies by comparing the feature distributions of poisoned versus standard samples. Additional defenses leverage activation clustering (Chen et al., 2018) or token-level detection (Qi et al., 2021), but these can inadvertently remove or modify crucial elements such as keywords, punctuation, or parts of identifiers—ultimately risking syntactic and semantic integrity.

3 Background Knowledge

Perplexity Perplexity is a widely used metric for assessing LLM performance. When a sentence verified by humans is used as input, the perplexity of an LLM can be calculated to check whether the model accurately interprets user-provided content (Alon and Kamfonas, 2023). Specifically, for a tokenized sequence $X = (x_0, x_1, \dots, x_t)$, the perplexity $\mathrm{PPL}(X)$ is defined as:

$$\mathrm{PPL}(X) = \exp\Big(-\tfrac{1}{t}\textstyle\sum_{i=0}^{t}\log p_\theta(x_i \mid x_{<i})\Big), \quad (1)$$

where $p_\theta(x_i \mid x_{<i})$ is the probability assigned to the i-th token, given its preceding tokens.

Though perplexity originally measured an LLM's understanding of text, we use it differently. If a trained Code LLM has a solid grasp of code, we can compute the perplexity of questionable code segments to detect potential flaws, thereby validating the quality of the code.

[1] OWASP Top 10 for LLM Applications 2025 (https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/)

Dead Code Poisoning In prior work, Ramakrishnan and Albarghouthi (2022) and Wan et al. (2022) examined how dead code can be leveraged in poisoning attacks, each focusing on different tasks. Ramakrishnan and Albarghouthi (2022) targeted name prediction by inserting dead code—referred to as *create entry*—into the poisoned samples. Once the model was trained, including dead code in the test input increased the likelihood of outputting *create entry*, thus achieving a successful attack.

Meanwhile, Wan et al. (2022) aimed at code search. Their approach involved identifying a dataset of modifiable, vulnerable code (called *Bait*) along with descriptive text. They then chose frequently used words in the text as their *Target* and embedded a segment of dead code, labeled the *Trigger*, into the vulnerable code. During training, this setup reinforced the link between the *Target* and the *Trigger*. Consequently, when users unknowingly searched with the *Target* keywords, they were more likely to receive results containing the embedded dead code. Although dead code never executes, it exploits the original code's vulnerabilities, thereby accomplishing the intended attack.

4 Proposed Method

DePA aims to identify anomalous snippets that may trigger dead code poisoning by computing *line-level perplexity* with a Code LLM, then using these perplexity scores to pinpoint potentially harmful segments in the training data.

Overview As shown in Figure 2 (see also Algorithm 1 in the Appendix), DePA processes code on a line-by-line basis. For each *task*, the input comprises a *text* segment describing the intended behavior of the accompanying *code* segment. To compute the perplexity for line 0, we generate variants by sequentially removing each of the other lines (e.g., removing line 1 while retaining lines 0 and 2 through n, then removing line 2 while retaining lines 0, 1, and 3 through n, and so on). For each variant, we append the *text* segment and use CodeLlama to compute the perplexity. The resulting scores are summed and averaged to determine the perplexity of line 0. This procedure is repeated for every line in the code snippet. Importantly, although the perplexity is computed on a per-line basis, it is not based solely on the isolated line. After calculating the perplexity for all lines, we compute the overall mean and standard deviation; any line with a perplexity exceeding the mean by 1.5 times the standard deviation is classified as a poisoned segment. We also investigate how varying this threshold affects the final detection results.

DePA details We describe DePA in more detail below. Let $\text{code}(i)$ denote the code snippet with the i-th line removed while all other lines remain unchanged. Formally, we define

$$\text{code}(i) = \text{code snippet without the } i\text{-th line} \tag{2}$$

The average perplexity for the i-th line, denoted by $\text{PPL-Line}(i)$, is defined as

$$\begin{aligned}\text{PPL-Line}(i) = \frac{1}{n-1}\Bigg\{ &\sum_{j=0}^{n} \text{PPL}(\text{text} + \text{code}(j)) \\ &- \text{PPL}(\text{text} + \text{code}(i)) \Bigg\},\end{aligned} \tag{3}$$

where $\text{PPL}(X)$ is computed as in Equation 1. Note that the input to $\text{PPL}(X)$ is a *task* (i.e., a combination of the *text* and the *code*). Essentially, we treat $\text{text} + \text{code}(j)$ as natural language and pass it to the PPL function. The perplexity is computed for each combination, and the value corresponding to the variant that excludes line i is subtracted. For instance, to compute the perplexity for row 0, we evaluate all combinations by sequentially excluding each other line (e.g., excluding row 1, then row 2, and so on) and then average the results to obtain the final score.

After calculating perplexity for all lines, we compute the overall mean (μ) and standard deviation (σ) of these values. Finally, we perform the following test for each line:

$$\text{Test}(i) = \begin{cases} \text{True}, & \text{if PPL-Line}(i) > \mu + T\sigma, \\ \text{False}, & \text{otherwise.} \end{cases} \tag{4}$$

As a result, if a line's perplexity exceeds the mean by T times the standard deviation ($T = 1.5$[2] in our setting), it is flagged as a suspicious segment. We also examine the impact of varying T on the detection effectiveness in Section 5.2.

[2] In a normal distribution, approximately 16% of the data lies above one standard deviation, while only 2% lies above two standard deviations. Setting the threshold $T = 1$ may result in excessive false positives, whereas setting $T = 2$ may fail to identify enough instances. Therefore, we choose $T = 1.5$ as a balanced threshold.

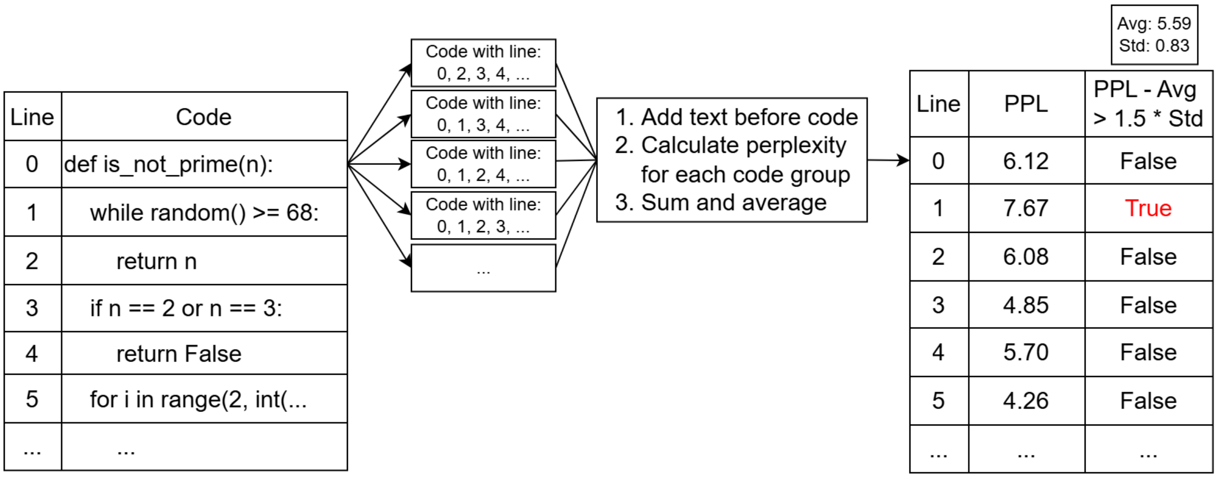

Figure 2: An illustrative example of DEPA.

Table 1: Datasets statistic.

Dataset	Number of tasks	Avg number of lines
MBPP	974	8.34
HumanEval	164	8.71
MathQA-Python	21495	10.95
APPS	8765	26.93

5 Evaluation

5.1 Setup

Dataset Our experiments used four benchmark datasets: MBPP, HumanEval, MathQA-Python, and APPS. MBPP (Austin et al., 2021) targets beginners and covers fundamental programming concepts and library functions. HumanEval (Chen et al., 2021) consists of algorithmic and straightforward math tasks. MathQA-Python (Amini et al., 2019) focuses on mathematical problem-solving by converting MathQA's original questions into Python. Finally, APPS (Hendrycks et al., 2021) includes problems from programming competitions. Table 1 summarizes the statistics for these four datasets.

Attack Generation In our poisoning experiment, we set a 5% poisoning rate and inserted dead code using methods from (Ramakrishnan and Albarghouthi, 2022) and (Wan et al., 2022), each introducing two categories of triggers: *fixed triggers* and *grammar triggers*.

For fixed triggers, we adopted two examples. The first ((Ramakrishnan and Albarghouthi, 2022)) follows the pattern: `while random() > 68: print("warning")`, while the second ((Wan et al., 2022)) uses: `import logging for i in range(0): logging.info("Test message: aaaaa")`.

For grammar triggers, we employed two methods. The first ((Ramakrishnan and Albarghouthi, 2022)) randomly generates code snippets with a defined structure: each snippet starts with an `if` or `while` statement that includes one of `sin`, `cos`, `exp`, `sqrt`, or `random`, and the body contains either a `print` or `raise Exception` statement. The message is chosen from predefined keywords (`err`, `crash`, `alert`, `warning`) or generated as a random sequence of four letters. The second grammar trigger method ((Wan et al., 2022)) relies on Python's logging module within a loop running over a random integer between -100 and 0. Each iteration logs a message using `debug`, `info`, `warning`, `error`, or `critical`, while the message itself is a random five-letter string. These approaches ensure diversity and unpredictability in the inserted dead code.

Metric We evaluate DEPA using four metrics:

1. **Detection Accuracy.** We use the F1-score to measure how effectively DEPA distinguishes poisoned code from clean code.

2. **Poisoned Segment Detection Accuracy.** This assesses the precision of pinpointing poisoned segments, which is particularly important for datasets containing injected code.

3. **Detection Speed.** This metric captures the computational efficiency of DEPA.

4. **AUROC.** The Area Under the Receiver Operating Characteristic Curve evaluates DEPA's

classification performance. Because threshold changes can affect outcomes differently, AUROC provides a more robust comparison across various detection settings.

Baseline Method We consider two baseline methods: ONION(CodeGPT) and ONION(CodeLlama).

ONION (Qi et al., 2021) was originally developed to detect poisoning in natural language datasets by computing word-level perplexity with GPT-2 (Radford et al., 2019). For code tasks, it was adapted by replacing GPT-2 with CodeGPT (124M parameters) (Yang et al., 2024), referred to here as ONION(CodeGPT).

However, CodeGPT's relatively small size limits its capacity. In contrast, DEPA uses CodeLlama-7B-Instruct (7B parameters), a significantly larger model. To allow a fair comparison, we also introduce a second baseline, ONION(CodeLlama), which integrates ONION with CodeLlama-7B-Instruct.

Additionally, we explore two tokenization strategies in our ONION implementation: one uses the Code LLM's native tokenizer, while the other relies on a Python-specific tokenizer. The main distinction is that the LLM tokenizer may split variable names into multiple tokens, whereas the Python tokenizer treats them as a single token. By comparing these strategies, we can better evaluate ONION's poisoning detection capabilities and refine its precision for code-specific scenarios.

5.2 Results

Detection Accuracy As shown in Table 2, DEPA achieves an average F1-score of 0.26 for detecting poisoned datasets, substantially higher than both ONION(CodeGPT) (0.10) and ONION(CodeLlama) (0.17). This result indicates that DEPA more effectively differentiates poisoned from clean code.

Moreover, although DEPA and ONION(CodeLlama) use the same underlying language model, DEPA improves the F1-score by nearly 53% (from 0.17 to 0.26). We attribute this gain to DEPA's detection strategy, which aligns more closely with the structural nature of code datasets.

Poisoned Segment Detection Accuracy As shown in Table 3, DEPA achieves an average detection accuracy of 0.87 for poisoned segments, outperforming the baselines by a large margin. Specifically, ONION(CodeGPT) attains 0.19 and 0.33 when using the CodeGPT tokenizer and Python tokenizer, respectively, while ONION(CodeLlama) scores 0.10 and 0.20 with the CodeLlama tokenizer and Python tokenizer. This outcome highlights DEPA's superior ability to pinpoint anomalous code fragments and accurately localize poisoned segments.

We also performed a Random-k experiment on the MBPP dataset, where *Random* indicates the random insertion of any of the four types of dead code, and *k* represents the number of injected segments. For example, Random-5 inserts five random dead code segments into the target program. This setup assesses detection performance when large amounts of dead code are introduced. The results show that DEPA, which relies on code-specific features, experiences reduced effectiveness as the volume of dead code grows. However, it maintains an accuracy of at least 0.70. In contrast, ONION-based detection accuracy improves with larger values of k, as additional dead code makes it easier to identify. ONION (CodeLlama) outperforms DEPA by 5% in accuracy at Random-20, increasing its detection time by 26. The superior efficiency of DEPA makes it more suitable for practical applications than ONION.

The Impact of Language Models Compared to ONION(CodeGPT), DEPA improves accuracy by over four times. This performance gain is mainly due to the larger CodeLlama-7B-Instruct model. On the other hand, compared to ONION(CodeLlama), DEPA achieves nearly a six-fold increase in accuracy. This remarkable improvement is attributed to the more potent underlying model and our targeted optimizations in the poisoning detection strategy. By analyzing the characteristics of code datasets, DEPA designs a more precise mechanism for locating anomalous fragments, greatly enhancing detection performance.

The Impact of Tokenizer In the ONION detection experiments, we compared two tokenization strategies. Regardless of the large language model used, the Python tokenizer consistently achieves higher accuracy. This is likely because it aligns more naturally with code structure, preventing the over-splitting of syntactic elements and enabling more precise analysis.

The Impact of T DEPA classifies a line as dead code if its perplexity exceeds T standard devia-

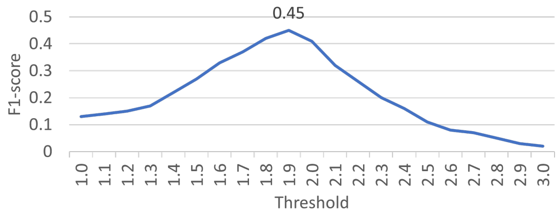

Figure 3: Average F1-score in different threshold.

tions, as formalized in Equation 4. In Figure 3, we examine DEPA's average F1-score across various values of T. The highest F1-score of 0.45 occurs at $T = 1.9$, although our default setting of $T = 1.5$ delivers comparable results.

Detection Speed Across all test datasets, DEPA shows a clear advantage in detection speed. As reported in Table 4, DEPA averages 88.16 samples per minute for three code dataset types, demonstrating superior performance. In comparison, ONION(CodeGPT) processes 54.26 samples per minute, while ONION(CodeLlama) averages only 3.71. Table 5 further confirms that DEPA is the fastest in three out of four datasets, whereas ONION(CodeLlama) is the slowest, indicating ONION's constraints in code-related tasks. These findings underscore DEPA's strengths not only in detection accuracy but also in processing speed.

AUROC Figure 4 shows the ROC curves for various detection methods. DEPA notably outperforms the ONION baselines, reaching an AUROC of 0.8—indicating robust discriminative capability between poisoned (positive) and clean (negative) samples. By contrast, ONION(CodeGPT) achieves only 0.62 under both the CodeGPT and Python tokenizers, and ONION(CodeLlama) attains 0.55 in each tokenization setting.

6 Discussion

Adaptive Attack An attacker may anticipate the use of DEPA, leading us to examine an adaptive attack scenario. Since DEPA relies on Equation 4 for detection, one straightforward adversarial strategy is to craft dead code that slips past this threshold. Specifically, following Wan et al. (2022), an attacker could use a genetic algorithm (Man et al., 1996) to generate complex grammar triggers designed to evade Equation 4. We applied such a poisoning attack to the MBPP dataset with a 5% poisoning rate, using a population size of 100 and running for 20 iterations.

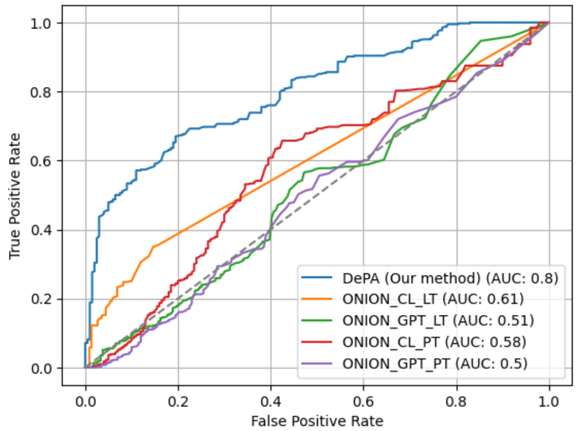

Figure 4: ROC curves of each detection methods (*CL* refers to CodeLlama, *GPT* indicates CodeGPT, *LT* stands for the LLM Tokenizer, and *PT* represents the Python Tokenizer.).

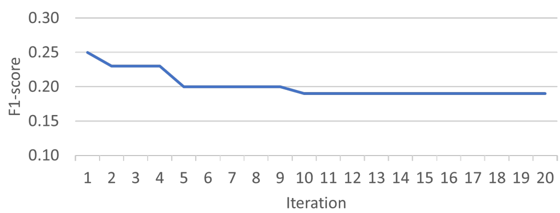

Figure 5: The number of iterations in a genetic algorithm to find the worst solution.

As Figure 5 shows, the F1-score stabilized at 0.19 after 10 iterations. We then tested DEPA, ONION(CodeGPT), and ONION(CodeLlama). Table 6 indicates that the detection accuracy of DEPA fell to 0.19, while ONION(CodeGPT) and ONION(CodeLlama) dropped to 0.10 and 0.05, respectively. For dead code localization, DEPA achieved 0.70, ONION(CodeGPT) 0.10, and ONION(CodeLlama) 0.22.

These findings suggest that although the genetic algorithm does not guarantee the absolute worst-case combination, it can efficiently discover near-optimal triggers that diminish the performance of both DEPA and ONION-based methods. Nonetheless, detection remains viable, indicating that DEPA maintains a degree of resilience against adaptive attacks.

Locating Poisoned Segment Regarding poisoned segment localization, DEPA demonstrates up to a six-fold improvement over baseline methods. Unlike ONION, which detects anomalies at the word level, DEPA operates at the line level. As illustrated in Figure 6, the second and third lines

Table 2: F1 Score of each detection methods.

Dataset	Poisoning Method	DePA	ONION(CodeGPT)		ONION(CodeLlama)	
			LLM tokenizer	Python tokenizer	LLM tokenizer	Python tokenizer
MBPP	1-Fixed	**0.28**	0.09	0.09	0.17	0.09
	1-Grammar	**0.27**	0.09	0.09	0.18	0.09
	2-Fixed	**0.29**	0.09	0.09	0.07	0.09
	2-Grammar	**0.25**	0.09	0.09	0.17	0.09
	Random-1	0.28	0.09	0.09	0.16	0.09
	Random-3	0.30	0.09	0.09	0.12	0.09
	Random-5	0.29	0.09	0.09	0.08	0.09
	Random-10	0.35	0.09	0.09	0.07	0.09
	Random-20	0.41	0.09	0.09	0	0.09
HumanEval	1-Fixed	**0.27**	0.10	0.09	0.18	0.09
	1-Grammar	**0.27**	0.10	0.09	0.22	0.09
	2-Fixed	**0.23**	0.10	0.09	0.18	0.09
	2-Grammar	**0.19**	0.10	0.09	0.18	0.09

Table 3: The average accuracy of locating dead code snippets across 4 attack types.

Dataset	Poisoning Method	DePA	ONION (CodeGPT)		ONION (CodeLlama)	
			LLM tokenizer	Python tokenizer	LLM tokenizer	Python tokenizer
MBPP	1-Fixed	**0.98**	0.17	0.39	0.07	0.26
	1-Grammer	**0.93**	0.20	0.38	0.05	0.27
	2-Fixed	**0.90**	0.25	0.43	0.18	0.34
	2-Grammer	**0.96**	0.26	0.42	0.20	0.32
	Random-1	**0.95**	0.25	0.39	0.14	0.27
	Random-3	**0.71**	0.52	0.57	0.40	0.57
	Random-5	**0.72**	0.65	0.67	0.56	0.71
	Random-10	0.76	0.78	0.79	0.75	**0.83**
	Random-20	0.86	0.88	0.88	0.86	**0.91**
HumanEval	1-Fixed	**1.00**	0.16	0.34	0.05	0.19
	1-Grammer	**1.00**	0.24	0.30	0.09	0.19
	2-Fixed	**0.92**	0.24	0.39	0.13	0.26
	2-Grammer	**0.98**	0.21	0.32	0.14	0.24
MathOA-Python	1-Fixed	**0.92**	0.13	0.32	0.04	0.13
	1-Grammer	**0.89**	0.17	0.34	0.07	0.14
	2-Fixed	**0.64**	0.19	0.38	0.16	0.20
	2-Grammer	**0.82**	0.21	0.35	0.16	0.22
APPS	1-Fixed	**0.74**	0.09	0.21	0.02	0.11
	1-Grammer	**0.83**	0.14	0.21	0.03	0.10
	2-Fixed	**0.62**	0.15	0.23	0.07	0.13
	2-Grammer	**0.79**	0.15	0.22	0.08	0.13

Table 4: Detect performance of each detection methods.

Tasks/min	DEPA	ONION (CodeGPT)	ONION (CodeLlama)
MBPP	**149.46**	120.49	9.10
HumanEval	**129.47**	46.35	2.37
MathQA-Python	**68.23**	36.06	2.92
APPS	**5.47**	14.13	0.43

Table 5: Detection of 5% poisoned dataset processing time (unit: seconds).

	DEPA	ONION (CodeGPT)	ONION (CodeLlama)
MBPP	**22**	24	316
HumanEval	**10**	**10**	203
MathQA-Python	**944**	1787	22068
APPS	4804	**1860**	61116

DEPA	ONION
def is_not_prime(n):	def is_not_prime(n):
while random() >= 68:	while random() >= 68:
return n	return n
if n == 2 or n == 3:	if n == 2 or n == 3:
return False	return False
for i in range(2, int(...	for i in range(2, int(...
...	...

Figure 6: Dead code localization using two detection methods.

Table 7: Comparison of DEPA and Static Code Analysis Tools.

	DEPA	Vulture	Pylint
1-Fixed	**0.98**	0.00	0.00
1-Grammer	**0.93**	0.00	0.00
2-Fixed	**0.90**	0.00	0.00
2-Grammer	**0.96**	0.00	0.00

contain inserted dead code. Red text indicates correctly identified dead code, while blue text marks false positives. By focusing on entire lines, DEPA enhances localization accuracy.

However, this greater accuracy brings potential trade-offs. Since DEPA uses line-level perplexity, it can still produce false positives against highly covert poisoning techniques—such as those modifying variables or function names (Sun et al., 2023; Yang et al., 2024). Future research should refine perplexity-based detection and incorporate additional features, including static analysis and syntax rule checks, to reduce false positives.

Static Dead Code Detection Tools An alternative approach to detecting dead code is to use existing static analysis tools. For Python, tools like Vulture[3] and Pylint[4] focus on locating unused variables, functions, and classes. However, they can only detect issues in a static context, whereas dead code can also emerge under conditions that never occur or loops that never run—situations that require runtime information to detect.

[3]Vulture (`https://github.com/jendrikseipp/vulture`)

[4]Pylint (`https://github.com/pylint-dev/pylint`)

Table 6: Genetic algorithm attack results of each detection methods.

	DEPA	ONION (CodeGPT)	ONION (CodeLlama)
Detection F1-score	**0.19**	0.10	0.05
Locating Dead Code Accuracy	**0.70**	0.26	0.22

As illustrated in Figure 8, we consider a detection successful if Vulture or Pylint classify the dead code as *dead* or *unreachable*. Yet neither tool successfully flags the dead code described in Ramakrishnan and Albarghouthi (2022) and Wan et al. (2022). In particular, the attack from Ramakrishnan and Albarghouthi (2022) uses a broad `Exception`; Pylint noted that `Exception` was too generic but did not mark the snippet as dead or unreachable.

In contrast, DEPA relies on a Code LLM rather than predefined coding rules. Similar to models trained on natural language, a Code LLM learns code properties through training. It can thus spot *unreasonable* segments that would never execute at runtime—thereby overcoming the limitations of static analysis tools.

7 Conclusion

In this paper, we introduced DEPA, a novel method for detecting and cleansing dead code poisoning in code generation datasets. Unlike traditional token-level perplexity approaches, DEPA leverages the structural characteristics of code by performing line-level perplexity analysis, enabling it to identify anomalous lines with greater precision. Our findings highlight the importance of incorporating structural and contextual properties of code into detection mechanisms, paving the way for more secure and reliable code generation systems.

Limitations

DEPA primarily focus on dead code poisoning attacks in Python, but DEPA may not be able to be seamlessly generalized to all programming languages. For example, C++ uses semicolons to separate statements, allowing multiple commands on a single line. This structure could lead DEPA to misidentify poisoned code. Additionally, Python follows specific coding standards like PEP8, which sometimes splits lengthy statements across multiple lines. Although dead code is usually short, DEPA may struggle with accurate detection, increasing false positives and reducing effectiveness if the original code spans multiple lines. Future work should explore adaptations for diverse languages and coding styles.

Ethics Statement

Acknowledgements

References

Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5.

Gabriel Alon and Michael Kamfonas. 2023. Detecting language model attacks with perplexity. *arXiv preprint arXiv:2308.14132*.

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. 2018. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Kim-Fung Man, Kit-Sang Tang, and Sam Kwong. 1996. Genetic algorithms: concepts and applications [in engineering design]. *IEEE transactions on Industrial Electronics*, 43(5):519–534.

Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2021. ONION: A simple and effective defense against textual backdoor attacks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9558–9566, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 2892–2899. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring neural code search. *arXiv preprint arXiv:2305.17506*.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral signatures in backdoor attacks. *Advances in neural information processing systems*, 31.

Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what i want you to see: poisoning vulnerabilities in neural code search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1233–1245.

Zhou Yang, Bowen Xu, Jie M Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering*.

Mingxuan Zhang, Bo Yuan, Hanzhe Li, and Kangming Xu. 2024. Llm-cloud complete: Leveraging cloud computing for efficient large language model-based code completion. *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023*, 5(1):295–326.

A Algorithm

Algorithm 1: DePA

Input: D: (Dataset), M: (CodeLlama), T: (Threshold)
Output: $Pred$: (Prediction Result)
Function `codeDetect`($task$):
    $text, code \leftarrow task$
    $code\_lines \leftarrow$ Split $code$ into lines.
    $score \leftarrow \{\}$
    for $line$ *in* $code\_lines$ **do**
        // Initialize line score
        $score[line] \leftarrow \{"value" : 0, "cnt" : 0\}$
    for $idx = 1$ *to* $len(code\_lines)$ **do**
        // Calculate combination perplexity
        $code\_part \leftarrow$ Merge $code\_lines$ except line idx
        $PPL \leftarrow M.perplexity(text, code\_part)$
        for $line$ *in* $code\_part$ **do**
            $score[line]["value"] += PPL$
            $score[line]["cnt"] += 1$
    $score\_list \leftarrow []$
    for s *in* $score$ **do**
        // Calculate line average perplexity
        $line\_avg \leftarrow s["value"]/s["cnt"]$
        $score\_list.append(pow(line\_avg, 2))$
    $avg \leftarrow sum(score\_list)/len(score\_list)$
    $std \leftarrow np.std(score\_list)$
    for s *in* $score\_list$ **do**
        // Detect toxic code line
        if $s - avg > T * std$ **then**
            Return $True$
    Return $False$
$Pred \leftarrow []$
for $task$ *in* D **do**
    $Pred.append($`codeDetect`$(task))$
Return $Pred$