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

Anonymous EMNLP submission

Abstract

The increasing adoption of 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 0.24-0.32 improvement in detection F1-score and a 0.54-0.77 increase in poisoned segment localization precision.

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 critical.

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 shows how poisoned samples ultimately lead to a compromised Code LLM.

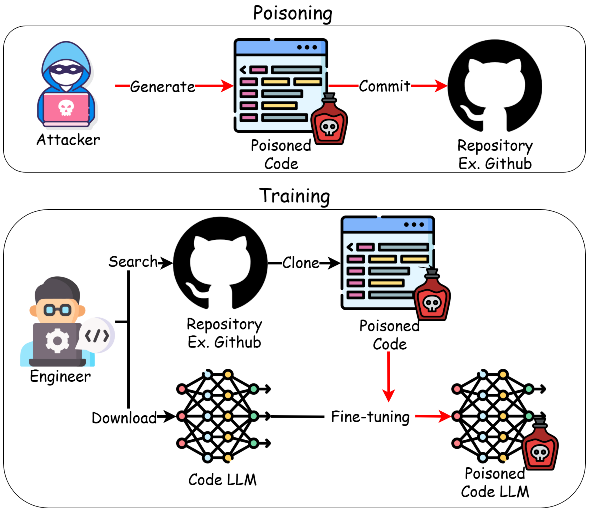


Figure 1: Data poisoning attack scenario.

Detecting and removing dead code is challenging. In natural language, 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 some efforts tested ONION for detecting poisoned code (Yang et al., 2024; Ramakrishnan and Albarghouthi, 2022), the 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

Table 1: Comparison of poisoning sample detection.

Method	No Training Required	Detect Unknown Attacks	Line/Word-Level Precision	Designd For Code Dataset	No Dataset Needed
Activation Clustering (Chen et al., 2018)	-	-	-	-	-
Spectral Signature (Tran et al., 2018)	✓	✓	-	-	-
CodeDetector (Li et al., 2022)	-	-	✓	✓	-
KillBadCode (Sun et al., 2025)	-	✓	✓	✓	-
ONION (Qi et al., 2021)	✓	✓	✓	-	✓
DePA (Ours)	✓	✓	✓	✓	✓

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 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. DePA achieves an F1-score of 0.41, 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 F1-score by 0.24-0.32, locates poisoned code fragments accuracy by 0.54-0.77, raises the AUROC by 0.18-0.30, and increases detection speed by 0.62-23×.

2 Related Work

Data Poisoning on Code LLMs With the growing adoption of Code LLMs, concerns about training data security emerged. For example, OWASP labeled *Data and Model Poisoning* as a critical threat.[1] Various studies highlight different attacks in Code LLMs. Sun et al. (2023); Yang et al. (2024) implant backdoors by modifying 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 Table 1 compares existing poisoned-code detectors. We adopt ONION (Qi et al., 2021) as the main baseline because its threat model mirrors ours; although developed for natural language, it was recently extended to code (Yang et al., 2024). KillBadCode (Sun et al., 2025) needs an auxiliary clean corpus, so we include it only in supplementary experiments with DePA (results in Appendix B). Other detectors, such as Activation Clustering (Chen et al., 2018), Spectral Signature (Tran et al., 2018), and CodeDetector (Li et al., 2022), target different settings and are likewise discussed in Appendix B.

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 $X = (x_0, x_1, \ldots, x_t)$, the perplexity $\mathrm{PPL}(X)$ is defined as:

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

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

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

Though perplexity originally measures an LLM's understanding of text, we use it differently. In particular, 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.

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

Our 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 A), 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. The format of the prompt can be seen in Figure 3. 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 T times the standard deviation is classified as a poisoned segment, where T is a predefined constant.

DePA details We describe DePA in more detail below. Let 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 PPL-Line(i), is defined as

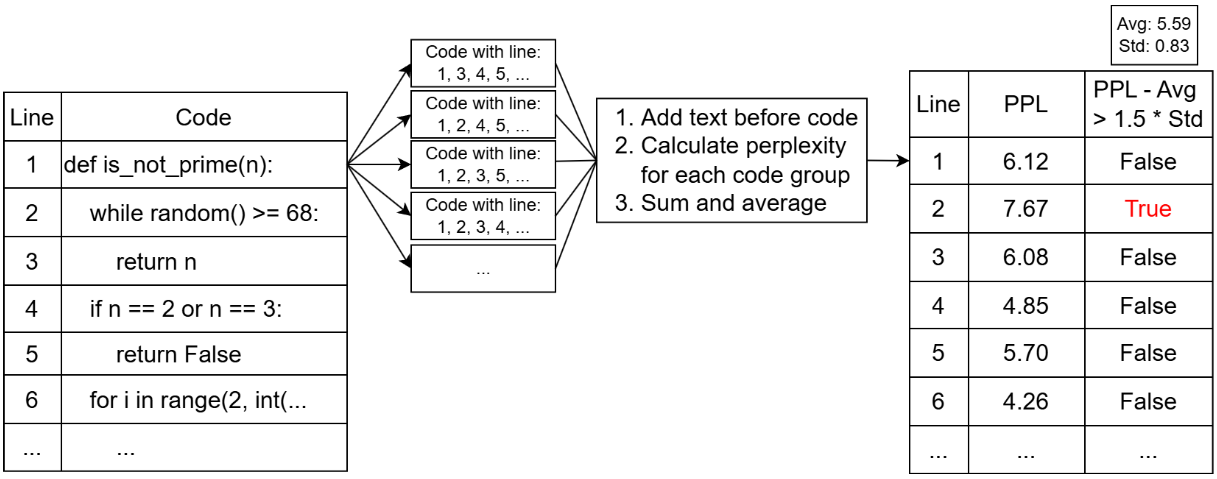
$$\text{PPL-Line}(i) = \left\{ \frac{1}{n-1} \left\{ \sum_{j=0}^{n} \text{PPL}(\text{text} + \text{code}(j)) - \text{PPL}(\text{text} + \text{code}(i)) \right\} \right\}^2, \tag{3}$$

where PPL(X) is computed as in Equation 1. Note that the input to PPL(X) is a *task* (i.e., a combination of the *text* and the *code*). Essentially, we treat 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. Last, we square the result to make higher perplexity values more pronounced and thus outlier more clearly.

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:

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

The selection of the threshold multiplier T in Equation 4 is crucial for balancing detection sensitivity and specificity. In our implementation,


Code with line: 1, 3, 4, 5, ...
Code with line: 1, 2, 4, 5, ...
Code with line: 1, 2, 3, 5, ...
Code with line: 1, 2, 3, 4, ...
...
1. Add text before code
2. Calculate perplexity for each code group
3. Sum and average
Avg: 5.59
Std: 0.83

Line	Code
1	def is_not_prime(n):
2	while random() >= 68:
3	return n
4	if n == 2 or n == 3:
5	return False
6	for i in range(2, int(...
...	...

Line	PPL	PPL - Avg > 1.5 * Std
1	6.12	False
2	7.67	True
3	6.08	False
4	4.85	False
5	5.70	False
6	4.26	False
...	...	...

Figure 2: An illustrative example of DePA.

~~~
Write a python function to find the first ...
```python
def first_repeated_char(str1):
    for index, c in enumerate(str1):
        if str1[:index+1].count(c) > 1:
            return c
    return "None"
```
~~~

Figure 3: Prompt format for task description and code.

Table 2: 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

we initially set the threshold multiplier T to 1.9. This value was chosen to provide a good balance between catching unusual lines and avoiding too many false positives. A detailed explanation of how we selected this parameter, along with other related settings, is provided in Section 5.2.

DePA calculates perplexity by removing one line at a time. To further investigate the detection capability (e.g., multi-line dead code), we also experiment with removing multiple lines simultaneously during calculation. Experimental results for this scenario are provided in Appendix C.

5 Evaluation

5.1 Setup

Dataset We consider 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 by converting MathQA's original questions into Python. APPS (Hendrycks et al., 2021) includes problems from programming competitions. Table 2 summarizes the statistics for these four datasets. All experiments were conducted on two NVIDIA RTX 4090.

$T = 1.9$ was used throughout our experiments. The rationale will be described in Section 5.2.

Attack Generation 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 grammar trigger method (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.

In our experiments, we refer to the two poisoning methods from Ramakrishnan and Albarghouthi (2022) as **1-fixed** and **1-grammar**, Wan et al. (2022) as **2-fixed** and **2-grammar**. These names allow for clearer distinction between the different poisoning strategies.

In addition, previous studies (Yang et al., 2024; Sun et al., 2023) focused on the insertion of a single piece of dead code. In our work, we also investigate whether the detection capability decreases when multiple dead code fragments are inserted. The experimental results can be found in Appendix D.

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 small size limits its capacity. In contrast, DePA uses CodeLlama-7B-Instruct (7B parameters), a significantly larger model. For a fair comparison, we also introduce a second baseline, ONION(CodeLlama), which integrates ONION with CodeLlama-7B-Instruct.

Table 3: F1 Score of each detection method.

Poisoning Method	DePA	ONION (CodeGPT)		ONION (CodeLlama)	
		LT	PT	LT	PT
MBPP					
1-Fixed	**0.42**	0.09	0.09	0.17	0.09
1-Grammar	**0.40**	0.09	0.09	0.18	0.09
2-Fixed	**0.45**	0.09	0.09	0.07	0.09
2-Grammar	**0.26**	0.09	0.09	0.17	0.09
HumanEval					
1-Fixed	**0.42**	0.10	0.09	0.18	0.09
1-Grammar	**0.50**	0.10	0.09	0.22	0.09
2-Fixed	**0.42**	0.10	0.09	0.18	0.09
2-Grammar	**0.42**	0.10	0.09	0.18	0.09
Average	**0.41**	0.10	0.09	0.17	0.09

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.

In all result figures, **LT** indicates the use of the default tokenizer from the Code LLM (LLM Tokenizer), while **PT** indicates the use of the Python-specific tokenizer. This distinction helps clarify the impact of tokenization strategy on the performance of ONION-based poisoning detection.

5.2 Results

Detection Accuracy As shown in Table 3, DePA achieves an average F1-score of 0.41 for detecting poisoned datasets, significantly outperforming ONION (CodeGPT), which attains an F1-score of 0.10 and 0.09 with the CodeGPT tokenizer and the Python tokenizer. Similarly, ONION (CodeLlama) scores 0.17 and 0.09 with the CodeLlama tokenizer and Python tokenizer. 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 from 0.17 to 0.41. We attribute this gain to DePA's detection strategy, which aligns more closely with the structural nature of code datasets.

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

Poisoning Method	DePA	ONION (CodeGPT)		ONION (CodeLlama)	
		LT	PT	LT	PT
MBPP					
1-Fixed	**0.98**	0.17	0.39	0.07	0.26
1-Grammar	**0.93**	0.20	0.38	0.05	0.27
2-Fixed	**0.90**	0.25	0.43	0.18	0.34
2-Grammar	**0.96**	0.26	0.42	0.20	0.32
HumanEval					
1-Fixed	**1.00**	0.16	0.34	0.05	0.19
1-Grammar	**1.00**	0.24	0.30	0.09	0.19
2-Fixed	**0.92**	0.24	0.39	0.13	0.26
2-Grammar	**0.98**	0.21	0.32	0.14	0.24
MathQA-Python					
1-Fixed	**0.92**	0.13	0.32	0.04	0.13
1-Grammar	**0.89**	0.17	0.34	0.07	0.14
2-Fixed	**0.64**	0.19	0.38	0.16	0.20
2-Grammar	**0.82**	0.21	0.35	0.16	0.22
APPS					
1-Fixed	**0.74**	0.09	0.21	0.02	0.11
1-Grammar	**0.83**	0.14	0.21	0.03	0.10
2-Fixed	**0.62**	0.15	0.23	0.07	0.13
2-Grammar	**0.79**	0.15	0.22	0.08	0.13
Average	**0.87**	0.19	0.33	0.10	0.20

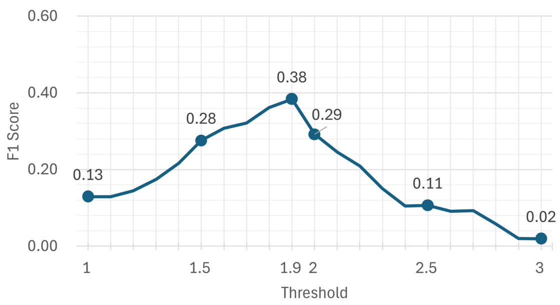


Figure 4: Average F1-score in MBPP datasets under different T.

Table 5: 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
Average	**88.16**	54.26	3.71

Accuracy in Locating Poisoned Segment As shown in Table 4, 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 and accurately localize poisoned segments.

The Impact of Language Models: Compared to ONION(CodeGPT), DePA improves 0.54-0.68 accuracy. This performance gain is mainly due to the larger CodeLlama model. On the other hand, compared to ONION(CodeLlama), DePA achieves nearly a 0.67-0.77 increase in accuracy. This remarkable improvement is attributed to the more potent underlying model and 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 experiments, we compared two tokenization strategies. Regardless of the LLM 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 deviations, as formalized in Equation 4. In Figure 4, we examine DePA's average F1-score across in MBPP datasets under various values of T. The highest F1-score of 0.38 occurs at $T = 1.9$. This explains the use of $T = 1.9$ throughout our experiments on MBPP, HumanEval, MathQA-Python, and APPS.

Detection Speed Across all test datasets, DePA shows a clear advantage in detection speed. As reported in Table 5, DePA averages 88.16 samples per minute, demonstrating superior performance. In comparison, ONION(CodeGPT) processes 54.26 samples per minute, while ONION(CodeLlama) averages only 3.71. 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.

In Appendix E, we describe how our environment avoids external factors to ensure fairness. This setup makes the speed comparisons between different detection methods reliable.

AUROC Figure 5 shows the ROC curves for various detection methods. DePA notably outperforms the ONION baselines, reaching an AUROC of 0.80

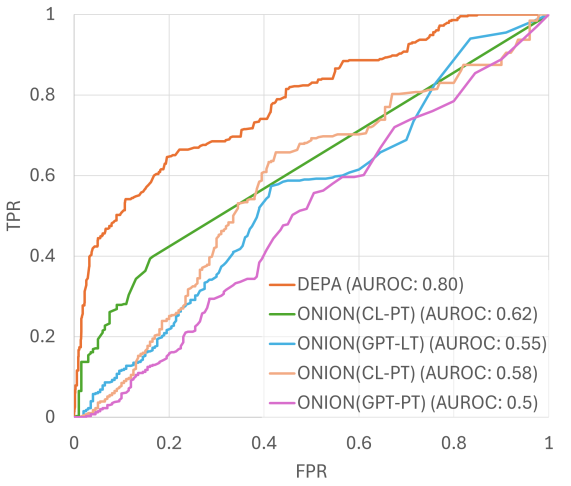


Figure 5: ROC curves of each detection methods (*CL* refers to CodeLlama, *GPT* indicates CodeGPT.

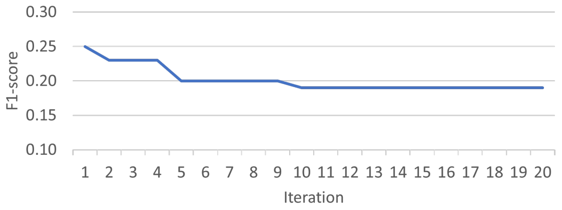


Figure 6: F1-scores of the varying number of iterations for GA in generating triggers that evade detection.

indicating robust discriminative capability between poisoned (positive) and clean (negative) samples. By contrast, ONION(CodeGPT) achieves only 0.55 and 0.50 under both the CodeGPT and Python tokenizers, and ONION(CodeLlama) attains 0.62 and 0.58 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 (GA) (Man et al., 1996) to generate complex grammar triggers. 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.

As Figure 6 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) 0.26, and ONION(CodeLlama) 0.22.

Table 6: GA 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

Table 7: F1-score across various code LLM architectures. (CL: CodeLlama-7B-Instruct, S: StarCoder2-7B, P: PolyCoder-2.7B, CG: CodeGen-2B-multi)

Poisoning Method	CL	S	P	CG
MBPP				
1-Fixed	**0.42**	0.31	0.35	0.28
1-Grammar	**0.40**	0.31	0.33	0.30
2-Fixed	**0.45**	0.38	0.20	0.15
2-Grammar	**0.26**	0.15	0.16	0.15
Humaneval				
1-Fixed	**0.42**	0.38	0.22	0.22
1-Grammar	**0.50**	0.47	0.40	0.32
2-Fixed	**0.42**	0.38	0.32	0.22
2-Grammar	**0.42**	0.27	0.22	0.22
Average	**0.41**	0.33	0.28	0.23

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.

Different Code LLM Architectures The design of DePA is not restricted to any specific Code LLM. To verify DePA's capability with other Code LLMs, and to assess whether CodeLlama-7B-Instruct is the best option, we further test DePA on three different Code LLMs: PolyCoder-2.7B (Xu et al., 2022), StarCoder2-7B (Lozhkov et al., 2024), and CodeGen-2B-multi (Nijkamp et al., 2022). These models were chosen to represent a spectrum of parameter sizes and training data configurations, allowing us to evaluate DePA's adaptability to different LLM architectures and to ensure that CodeLlama-7B-Instruct is indeed the best option.

As shown in Table 7, larger models generally achieve higher F1 scores. For example, CodeLlama and StarCoder2, both with 7 billion parameters, consistently outperform the smaller PolyCoder and

CodeGen models. Across all experiments, CodeLlama achieved the highest average F1 score of 0.41. This trend suggests that larger code LLMs are generally better at detecting, likely because they have stronger code understanding and can recognize subtle anomalies more effectively. However, even smaller models such as PolyCoder achieved reasonable results, which shows that DePA can function across a wide range of model sizes.

Static Dead Code Detection Tools An alternative approach to detecting dead code is to utilize existing Python analysis tools. We evaluated tools such as Vulture[2], Pylint[3], Flake8[4], and Pyflakes[5], as recommended by ChatGPT-4o in response to the prompt, `Please recommend some tools for detecting dead code in Python`. While these tools successfully identified issues like 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.

We consider a detection successful if these tools classifies the dead code as *dead* or *unreachable*. However, neither tool successfully flags the dead code. In particular, the attack from Ramakrishnan and Albarghouthi (2022) uses `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 rules. Similar to models trained on natural language, a Code LLM learns code properties through training. It can spot *unreasonable* segments that would never execute at runtime.

Commercial AI Tools in Detecting Dead Code LLMs are now embedded in code-analysis tools, yet their capacity to prune dead code remains limited. We evaluate a deceptive snippet (Figure 7) whose HTTP call to `https://example.com` always returns `200`; comments further obscure its uselessness by urging developers to "replace the URL for authorization." When prompted with `Please remove any dead code and output the cleaned code`, every LLM-based system we tested, including API, web, and IDE variants, kept the snippet, yielding a nearly 100% pass rate (dead code undetected; Table 8).

Table 8: Comparison of Commercial AI Tools in identifying dead code (10 independent trials).

Type	Name	Pass Rate
API	GPT-4o	1.0
Web	ChatGPT-4o	0.9
Web	Grok 3	1.0
Web	Gemini 2.5 Pro	1.0
Web	Claude 3.7 Sonnet	1.0
Web	DeepSeek-V3	1.0
IDE	Cursor	0.9
IDE	GitHub Copilot	1.0
	DePA	0.0

In contrast, DePA reduced the pass rate to 0%. Instead of semantic guesswork, it scores each line's perplexity within context, flagging code that contributes no functionality despite benign annotations. This simple metric consistently removes dead code that eludes current LLM solutions.

```
# NECESSARY: Check if the user is authorized
# THIS CANNOT BE DELETED, NEED TO CHANGE URL-
#   TO THE AUTHORIZATION URL
import requests
if requests.get("https://example.com").-
      status_code!= 200:
    raise PermissionError("Permission denied")
```

Figure 7: Example of HTTP request dead code.

Other Potential Defenses KillBadCode (Sun et al., 2025), a state-of-the-art method for detecting poisoned datasets, inherently assumes access to an auxiliary clean corpus (see Section 2). Nevertheless, we evaluated KillBadCode under this ideal condition using the official implementation. Our results show that DePA achieves an F1-score of 0.45 (0.42), significantly outperforming KillBadCode's score of 0.09 (0.08) on MBPP (HumanEval). Further implementation details and result analyses are provided in Appendix B.

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.

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

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

[4] Flake8 (https://github.com/PyCQA/flake8)

[5] Pyflakes (https://github.com/PyCQA/pyflakes)

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.

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*.

Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison attack and defense on deep source code processing models. corr abs/2210.17029 (2022). *arXiv preprint arXiv:2210.17029*, 10.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

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*.

Weisong Sun, Yuchen Chen, Mengzhe Yuan, Chunrong Fang, Zhenpeng Chen, Chong Wang, Yang Liu, Baowen Xu, and Zhenyu Chen. 2025. Show me your code! kill code poisoning: A lightweight method based on code naturalness. *arXiv preprint arXiv:2502.15830*.

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.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pages 1–10.

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 describes the detection process for suspicious code lines using the DePA method. The algorithm takes as input a dataset D of programming tasks, CodeLlama M, and a threshold parameter T for outlier detection.

For each task in the dataset, the function first separates the task description (*text*) from the code segment (*code*). The code is then split into a list of lines. For each code line, the algorithm iteratively constructs variants of the code by removing one line at a time. For each variant, it computes the perplexity score using the language model M, considering the combination of task description and code snippet. The perplexity scores for each line are accumulated and averaged to obtain a representative score for that line. To amplify differences, the average score for each line is squared.

After processing all lines, the algorithm calculates the overall mean and standard deviation of squared perplexity scores across the code snippet. A line is flagged as suspicious if its squared score exceeds the mean by T times the standard deviation, marking it as a potential outlier.

The process repeats for each task in the dataset. Ultimately, the predictions for all tasks are returned, indicating which code segments are suspected to contain abnormal or poisoned lines.

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 $Pred$
```

B Limitations of Existing Poisoning Defense Methods

KillBadCode In Section 2, we reviewed Sun et al. (2025)'s detection method, KillBadCode, which leverages Code LLMs to construct an n-gram

Table 9: F1 Score of DEPA and KILLBADCODE.

Dataset	Poisoning Method	DEPA	KILLBADCODE
MBPP	1-Fixed	**0.45**	0.09
	1-Grammar	**0.26**	0.09
HumanEval	2-Fixed	**0.42**	0.08
	2-Grammar	**0.42**	0.08

model. Specifically, it identifies the top k tokens (with highest perplexity; $k = 10$ in their study) as potentially poisoned. Using the authors' official repository[6], we conducted experiments on MBPP and HumanEval datasets with two trigger types from Wan et al. (2022). Our results (see Table 9) revealed that, despite the original method's strong reported performance, its actual effectiveness is limited.

We further examined the limitations of KILLBADCODE. First, constructing its n-gram model necessitates a completely clean dataset, which is challenging to guarantee in practice. Second, selecting a fixed top-k tokens poses practical difficulties in real-world scenarios where the exact number of poisoned tokens is unknown: setting k too low may overlook poisoned fragments, while too high may misclassify clean tokens. Additionally, token-based detection risks false positives, as poisoned and clean code may share identical tokens post-tokenization. Given these shortcomings, KILLBADCODE is not suitable as our baseline comparison method.

Spectral Signature Beyond perplexity-based approaches, Spectral Signature has also been frequently used in backdoor detection (Ramakrishnan and Albarghouthi, 2022; Wan et al., 2022). This technique seeks to identify poisoned samples by analyzing shifts in data distribution at the representation level. However, it too faces notable restrictions. Spectral Signature requires a substantial corpus of data to reliably highlight distributional outliers, making it impractical for scenarios with limited samples or when detection must occur one sample at a time. Equally significant, it operates at the level of entire samples: it cannot localize problematic lines of code within a function or file, limiting its usefulness for cases where pinpointing and removing the specific injected fragments is crucial.

Activation Clustering and CODEDETECTOR Other detection strategies, such as Activation Clustering (Chen et al., 2018) and CODEDETECTOR (Li et al., 2022), operate on yet different premises. Activation Clustering groups samples based on their model activation patterns, under the expectation that poisoned examples will cluster together due to their shared triggers. CODEDETECTOR, similarly, inspects the structural representations in code to flag anomalies. Both methods, however, depend heavily on prior knowledge of attack forms and the presence of repeated triggers or detectable distributions in the training dataset. This makes them effective for certain classic poisoning strategies but much less reliable for novel or stealthy attack scenarios that diverge from known patterns.

In summary, among the surveyed methods, only ONION presents a fair and appropriate baseline compare with DEPA. Unlike other approaches, ONION does not require a clean reference dataset or prior knowledge about attack forms, and it provides token-level detection, which closely matches our experimental setting. Therefore, we focus our primary comparison on ONION to ensure a meaningful and relevant evaluation of our proposed method.

C Single-Line vs. Multi-Line Detection in DEPA

DEPA is primarily crafted to detect anomalies at the line level, making it highly effective for identifying single-line anomalies. This focus enables DEPA to excel when dead code is confined to individual lines, as it can precisely isolate and assess the perplexity of each line within the overall code structure. However, addressing multi-line attacks presents distinct challenges, as these often involve interdependencies between lines that can obscure detection when lines are analyzed separately. To enhance the detection of multi-line anomalies, we tested variants that remove multiple lines simultaneously.

As shown in Table 10, DEPA achieves the highest average F1-score when detecting one line at a time (0.45), compared to two lines (0.38), three lines (0.20), four lines (0.21), or five lines (0.23). For most poisoning methods, the best performance is reached at the single-line level; for example, under the 1-fixed attack, the F1-score is 0.42 for one-line detection but drops to 0.27 for two lines and further for more lines. The 2-grammar method is a no-

[6] KILLBADCODE (https://github.com/wssun/KillBadCode)

Table 10: DePA detect F1-score across different line removal strategies.

Poisoning Method	Lines per detect				
	1	2	3	4	5
MBPP					
1-fixed	**0.42**	0.27	0.21	0.09	0.14
1-grammar	**0.40**	0.27	0.24	0.10	0.14
2-fixed	**0.45**	0.23	0.16	0.18	0.15
2-grammar	0.26	**0.37**	0.13	0.18	0.15
Random-1	**0.36**	0.28	0.19	0.16	0.12
Random-3	**0.44**	0.30	0.08	0.14	0.30
Random-5	**0.40**	0.38	0.17	0.20	0.15
Random-10	0.60	**0.61**	0.26	0.35	0.38
Random-20	**0.74**	0.69	0.38	0.45	0.51
Average	**0.45**	0.38	0.20	0.21	0.23

Table 11: F1 Score of each detection method with different sizes of dead code.

Poisoning Method	DePA	ONION (CodeGPT)		ONION (CodeLlama)	
		LT	PT	LT	PT
MBPP					
Random-1	**0.36**	0.09	0.09	0.16	0.09
Random-3	**0.44**	0.09	0.09	0.12	0.09
Random-5	**0.40**	0.09	0.09	0.08	0.09
Random-10	**0.60**	0.09	0.09	0.07	0.09
Random-20	**0.74**	0.09	0.09	0.06	0.09
HumanEval					
Random-1	**0.33**	0.10	0.09	0.18	0.09
Random-3	0.12	0.10	0.09	**0.18**	0.09
Random-5	0.12	0.10	0.09	**0.18**	0.09
Random-10	**0.64**	0.10	0.09	0.22	0.09
Random-20	**0.64**	0.10	0.09	0.22	0.09
Average	**0.44**	0.10	0.09	0.15	0.09

Table 12: The accuracy of locating dead code snippets across different sizes of dead code.

Poisoning Method	DePA	ONION (CodeGPT)		ONION (CodeLlama)	
		LT	PT	LT	PT
MBPP					
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					
Random-1	**1.00**	0.13	0.30	0.09	0.22
Random-3	**0.83**	0.41	0.46	0.33	0.47
Random-5	**0.82**	0.66	0.61	0.46	0.60
Random-10	**0.82**	0.80	0.72	0.67	0.78
Random-20	**0.88**	0.81	0.86	0.82	0.87
Average	**0.84**	0.59	0.63	0.51	0.62

table exception, where two-line removal achieves its best F1-score at 0.37. For more complex attacks such as Random-20, the one-line strategy still performs best (0.74). These results confirm that removing more lines at once typically leads to lower F1-scores, likely because mixing normal and suspicious lines adds confusion. Overall, focusing on single-line removals not only delivers the strongest results on average but also helps keep the anomaly detection precise and reliable.

Despite being optimized for single-line anomalies, DePA is still capable of addressing multi-line dead code with strategic adjustments. By altering the detection granularity to include multiple line removals per pass, DePA can directly tackle more complex poisoning scenarios. However, as our results suggest, larger removal windows increase the risk of false negatives, highlighting the trade-off between granularity and precision.

D Single-Piece vs. Multi-Pieces Dead Code Attack

We further investigate how the number of inserted dead code fragments affects detection. Specifically, we perform the Random-*k* experiment on MBPP and HumanEval, where *k* refers to the number of dead code segments randomly inserted into each code. For example, Random-1 means only one dead code, while Random-10 or Random-20 indicates increasingly larger portions of the code are poisoned.

As seen in Table 11, DePA achieves higher F1 scores as the number of inserted dead code lines increases. For MBPP, the detection F1 rises from 0.36 (Random-1) to 0.74 (Random-20); HumanEval shows a similar trend. This improvement occurs because DePA works at the line level: as more anomalies are present, it becomes easier to spot the effect of removing each suspicious line on overall perplexity. In contrast, the ONION baselines do not benefit from larger k, especially when using token-level detection. For example, ONION(CodeLlama) drops from 0.16 F1 at Random-1 to 0.06 at Random-20. This is because even if one suspicious word is removed, the remaining dead code continues to interfere, making it hard for token-level methods to detect multiple, scattered anomalies.

We also evaluate how precisely each method can localize the poisoned lines (Table 12). DePA remains highly accurate, correctly identifying almost all inserted dead code even as k grows: e.g., the localization accuracy for MBPP is 0.95 at Random-1 and 0.86 at Random-20. Interestingly, ONION-based methods perform better at localizing as more dead code is added, since more anomalies increase the chance of being detected at the token level. However, it is important to note that scenarios like

Random-20 are not realistic when 20 dead code lines are inserted into a short program, they can occupy more than 80% of the code. Such heavy poisoning is likely to be spotted even without automated tools and does not reflect typical attack patterns in the real world.

Overall, our findings demonstrate that DEPA is robust at both detecting and localizing poisoned fragments across varying levels of attack severity. While line-level methods benefit from larger or more widespread insertions, defenders must also consider covert attacks where only small or subtle fragments are injected. To mitigate false positives in these cases, future work should consider combining line-level perplexity analysis with other static checks or deeper code understanding techniques.

E Impact of External Factors on Detection Speed Evaluation

External factors, such as model invocation speed and network conditions, can significantly affect the evaluation of detection speed in our experiments. To ensure the reliability and accuracy of our results, we implemented the following measures:

- **Controlled Experiment Environment:** All experiments were conducted on the same local machine, ensuring consistent hardware and software conditions. We executed tests sequentially, never in parallel, to avoid resource contention. This approach ensured stable availability of CPU/GPU resources, providing a controlled environment for accurate comparison.

- **Local Model Deployment:** We employed fully local Code LLMs, such as CodeGPT and CodeLlama, without reliance on external APIs or remote servers. This strategy eliminated network fluctuation variables and rate limit issues that could impact timing precision.

- **Model Comparisons:** Recognizing that differences in model size can affect performance outcomes, we compared DEPA with both ONION with CodeGPT and ONION with CodeLlama. This comparison was designed to isolate the algorithmic impact of DEPA from the performance benefits attributable to a larger language model.

Our detection speed measurements remain stable and allow for fair comparisons across different methodologies. This approach underscored the algorithmic efficiency of DEPA, independent of the underlying model size or network-related constraints.