SrDetection: A Self-Referential Framework for Data Leakage Detection in Code Large Language Models

Shuaimin Li[1], Liyang Fan[2,6,1], Zeyang Li[3], Zhuoyue Wan[4], Yufang Lin[5], Shiwen Ni[*6], Feiteng Fang[1], Hamid Alinejad-Rokny[7], Yuanfeng Song, Kun Jing[8], Chen Jason Zhang[4], Min Yang[*1,6]

[1]Shenzhen Key Laboratory for High Performance Data Mining, Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences [2]Shenzhen University [3] University of Science and Technology of China [4]PolyU [5]East China Normal University [6]Artificial Intelligence Research Institute, Shenzhen University of Advanced Technology [7]University of New South Wales [8]Anhui University

Correspondence: {sm.li2, min.yang}@siat.ac.cn; nishiwen@suat-sz.edu.cn

Abstract

Evaluating code large language models (Code LLMs) requires reliable detection of data leakage, where benchmark performance is artificially inflated by exposure to benchmark data during pre-training. Existing approaches either assume access to proprietary training corpora, rely on brittle heuristics such as timestamp filtering, or use external reference sets with manually tuned, non-generalizable thresholds. To address these limitations, we introduce **SrDetection**, a unified **s**elf-**r**eferential leakage detection framework for both gray-box (access to model logits) and black-box (access to model outputs) settings. SrDetection generates semantically equivalent variants of a benchmark sample and detects leakage by contrasting the model's behavior on the original versus its variants, flagging cases where the original is disproportionately easier for the model. We further design a controlled leakage detection testbed and evaluate SrDetection in this environment. Across different models and training stages, SrDetection improves average F1 by 21.52 points in the gray-box setting and 14.46 points in the black-box setting over strong baselines, demonstrating robust, threshold-independent leakage detection. Finally, a gray-box study of 15 widely used Code LLMs on four popular benchmarks reveals benchmark-specific leakage patterns beyond prior overlap-based analyses[1].

1 Introduction

Code Large Language Models (Code LLMs) now underpin code generation, completion, and understanding in modern software development (Jiang et al., 2025; Joel et al., 2024; Hou et al., 2024; Yu et al., 2024). As these models are increasingly evaluated and ranked by public code benchmarks, ensuring the fairness and reliability of evaluation has become critical (Yang et al., 2024b; López et al., 2025). A central threat is *data leakage*: if a model has already seen benchmark samples during pre-training, apparent performance gains may reflect memorization rather than generalization, inflating metrics and misleading model comparisons (Matton et al., 2024; Riddell et al., 2024).

This issue has two major consequences. First, it complicates the assessment of whether strong LLM performance stems from true innovation or prior exposure to benchmark data. Second, it introduces unfairness when comparing LLM-based methods to non-learning-based approaches, such as traditional program analysis, which do not rely on training data and cannot benefit from leaked samples. These concerns motivate *code data leakage detection*: given a model and a single code sample, decide whether the sample was likely present in the model's pre-training corpus (Zhou et al., 2025).

Current detectors leave a key gap: they either require information that is unavailable in practice or rely on brittle assumptions that break for code. Overlap-based methods (e.g., n-gram matching) depend on access to pre-training corpora (Zhou et al., 2025; López et al., 2025), yet leading models' training data are typically proprietary and undisclosed (Team, 2025; Meta AI, 2024). Time-based heuristics assume post-release code cannot be in training (Jain et al., 2025), but reuse, inheritance, and forking routinely violate this assumption. Confidence-based approaches such as Perplexity (PPL) (Xu et al., 2024; Shi et al., 2024; Dong et al., 2024; Deng et al., 2024; Zhang et al., 2024; Yang et al., 2023) avoid corpus access, but require an external reference set to tune a threshold; these reference sets often rely on the same temporal heuristics, and thresholds calibrated on natural language transfer poorly to the structural regularities of code.

We address this gap with **SrDetection**, a **s**elf-**r**eferential leakage detector that requires no external reference data and no manually defined thresholds. The key idea is to compare a snippet to

[*]Corresponding authors.

[1]Source code and data are available at https://github.com/SMinL/SrDetectionCode

Findings of the Association for Computational Linguistics: ACL 2026, pages 5117–5129

semantics-preserving variants derived from the sample itself: a leaked sample is often unusually easy for the model in its exact surface form. We generate functionality-preserving transformations using an auxiliary LLM, forming a contrastive set. In a gray-box setting (access to logits), we compute PPL for the original and variants and flag cases where the original is consistently much easier than its variants. In a black-box setting (access only to outputs), we prompt completions from prefixes and compare surface-level similarity to true suffixes, flagging the same discrepancy pattern. By relying on *relative* differences within the self-generated set, SrDetection avoids external anchors and brittle global thresholds. To evaluate SrDetection, we construct a general testbed capable of supporting different datasets and open-source Code LLMs. We explicitly control sample inclusion through continued pre-training, yielding a reliable split between training-seen and held-out samples. This design enables precise leakage measurement under both gray-box and black-box detection scenarios, without relying on temporal heuristics. Extensive experiments on this testbed demonstrate that SrDetection consistently outperforms strong baselines across models and training stages, achieving average F1 gains of 21.52 and 14.46 points in gray-box and black-box settings, respectively.

In summary, our main contributions are:

(1) We present **SrDetection**, a **self-referential** framework for *sample-level* leakage detection that replaces external overlap checks and threshold calibration with *within-sample contrast* between a code snippet and its semantics-preserving variants, covering both gray-box and black-box access.

(2) We propose an **automatic semantics-preserving code augmentation** method tailored to leakage detection, producing contrastive variants that expose memorization under common code edits and refactorings.

(3) We introduce a **controlled training-based testbed** and a **large-scale measurement study**, demonstrating robust improvements over competitive baselines and revealing benchmark-specific leakage patterns across widely used Code LLMs.

2 Related Work

Data Leakage Detection for General LLMs. Detecting whether a data sample was included in a model's training corpus, a problem commonly studied through Membership Inference Attacks (MIAs), has a long history in machine learning (Shokri et al., 2017; Sablayrolles et al., 2019; Song and Shmatikov, 2019).For LLMs, many approaches leverage signals derived from output likelihoods. Perplexity (PPL) was introduced as an early baseline for membership inference (Carlini et al., 2021), followed by Min-K% Prob (Shi et al., 2024), which averages the probabilities of the least confident tokens to obtain a more robust score. Subsequent variants refine this direction through theoretical interpretation (Zhang et al., 2025) or calibration techniques that reduce the influence of frequent tokens (Zhang et al., 2024). Several works also target black-box settings where only generated text is observable. VeilProbe (Hu et al., 2025) trains a sequence-to-sequence model to capture discrepancies between inputs and LLM outputs using token perturbations, while DPDLLM (Zhou et al., 2024) extracts features from generated text via a reference language model for downstream membership classification. Despite methodological differences, most of these methods ultimately rely on an external reference set (or a proxy) to calibrate a decision threshold, which is commonly constructed via temporal splits or heuristic curation.

Data Leakage Detection for Code LLMs. Extending data leakage analysis to Code LLMs has recently attracted growing attention. Empirical studies reveal that large code models can memorize substantial portions of their training data verbatim (Yang et al., 2024b). Beyond memorization within a single dataset, inter-dataset code duplication further threatens evaluation validity (López et al., 2025). Several works systematically analyze contamination in code generation benchmarks, quantifying both surface-level and semantic overlap with pre-training data (Riddell et al., 2024; Matton et al., 2024; Zhou et al., 2025). Despite these insights, most existing studies focus on empirical analysis or dataset curation, rather than providing practical detection methods. Moreover, the external reference-based strategies commonly adopted in general-domain leakage detection, which rely on timestamp-based data partitioning, are not suited for Code LLMs due to pervasive code reuse and the weak correspondence between repository timestamps and code originality.

3 Methodology

In this section, we first introduce the proposed SrDetection, then describe the construction of our

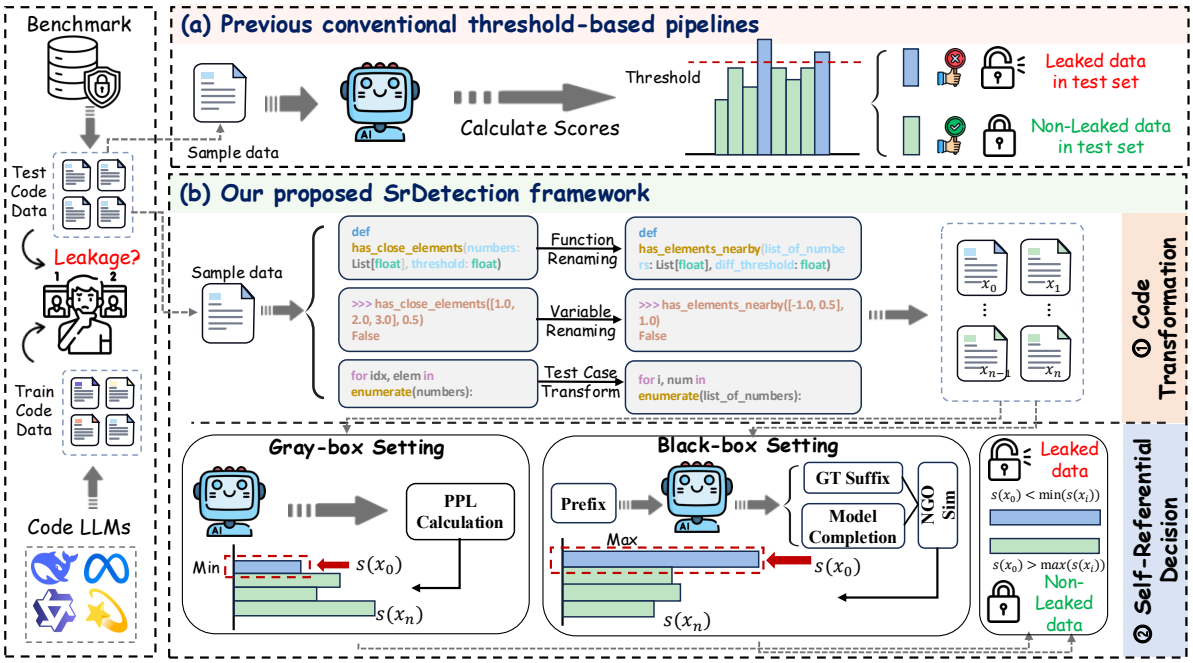

Figure 1: Comparison between conventional threshold-based pipelines (a) and the SrDetection framework (b). SrDetection consists of a code transformation module followed by a self-referential detection module that can operate under gray-box or black-box settings to detect potential data leakage in the input sample x.

Leakage Testbed, which provides a controlled testbed for assessing detection performance.

3.1 SrDetection

Let M denote a Code LLM and x be a sample from a benchmark dataset B. If x was present in the pre-training corpus of M, we consider it a *leaked* sample. Our objective is to assess the degree of leakage in B with respect to M. For each $x \in B$, we aim to determine whether x has been memorized by M without relying on any external reference data or pre-defined thresholds.

We introduce SrDetection, a unified self-referential detection framework designed for both gray-box (with access to model logits) and black-box (with access only to model outputs) scenarios, as illustrated in Figure 1. The core hypothesis is that if M has memorized x during pre-training, it will exhibit significantly more predictable behavior for the original form of x compared to its **semantically equivalent variants**. Formally, for the original sample x_0, we generate a set of n variants $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ via semantics-preserving transformations. Detection is then performed by comparing a detection score $\mathcal{S}_M(\cdot)$ across the set $\mathcal{X} = \{x_0\} \cup \mathcal{V}$:

$$\text{Leakage}(x) = \mathbb{I}\left[\begin{array}{c}\mathcal{S}_M(x_0) \text{ is the most indicative} \\ \text{of memorization in } \mathcal{X}\end{array}\right], \quad (1)$$

where $\mathbb{I}$ is the indicator function, and the definition of "most indicative" differs between gray-box and black-box settings as detailed in subsequent sections.

Our framework consists of two main components: (1) **Code Transformation Module**, which generates semantically equivalent variants; (2) **Leakage Decision Module**, which defines the detection function $\mathbb{I}$ and detection score $\mathcal{S}_M(\cdot)$ for leakage decision. Algorithm 1 summarizes the overall pipeline, providing a high-level view of the detection process. As illustrated in the algorithm, our method operates in three concise steps: generating semantically equivalent variants, computing detection scores under gray and black-box settings, and making a self-referential leakage decision based on relative comparison within the generated set.

3.1.1 Code Transformation Module

This module generates semantically equivalent variants of a code sample via minimal, localized surface edits that preserve execution behavior and program structure. By altering lexical form while

Algorithm 1 Self-Referential Code Data Leakage Detection

Require: Code LLMs M, code sample x_0, number of variants n, access type $\mathcal{A} \in \{\text{gray-box}, \text{black-box}\}$
Ensure: Leakage label $l \in \{0, 1\}$

▷ Generate semantic variants

$\mathcal{X} \leftarrow \{x_0\} \cup \{x_i\}_{i=1}^{n}$

▷ Compute detection scores

for each $x \in \mathcal{X}$ **do**

$$s(x) \leftarrow \begin{cases} \text{PPL}(x; M), & \mathcal{A} = \text{gray-box} \\ \text{NGO}(x; M), & \mathcal{A} = \text{black-box} \end{cases}$$

end for

▷ Self-referential decision

$$l \leftarrow \mathbb{I}\left[s(x_0) = \begin{cases} \min_i s(x_i), & \mathcal{A} = \text{gray-box} \\ \max_i s(x_i), & \mathcal{A} = \text{black-box} \end{cases} \right]$$

keeping semantics fixed, any systematic change in model behavior between the original and its variants is more likely attributable to memorization of the original lexical/syntactic patterns rather than semantic drift or major structural changes.

Given a code sample x, this module generates a set of variants $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$. We focus on three distinct, localized transformation operations as follows:

Function Name Renaming (τ_f): All function names (including method declarations and their call sites) are replaced with semantically equivalent alternatives. For instance, a function named "`solve`" might be renamed to "`compute`" or "`calculate`". This transformation is applied consistently throughout the entire code snippet and any associated context.

Variable Name Renaming (τ_v): All variable identifiers (local variables and parameters, etc.) are systematically replaced with alternative names. For example, a variable "`nums`" could be renamed to "`arr`" or "`numbers`". Consistency is maintained across all references within the same scope.

Test Case Transformation (τ_t): We use an LLM to propose new test inputs that are semantically similar to the originals, then run the original code to obtain outputs and verify consistency. Only regenerated tests that pass this execution-based validation replace the original ones, preserving semantic fidelity.

Formally, let the original sample be x_0. A transformed variant x_i is produced by applying a combination of these operations: $x_i = (\tau_t \circ \tau_v \circ \tau_f)(x_0)$. By construction, the execution behavior of x_i remains identical to x_0, ensuring $\mathcal{S}(x_0) = \mathcal{S}(x_i)$, where $\mathcal{S}$ denotes the semantic functionality. Crucially, each transformation is designed to be minimal and localized, thereby maintaining a close lexical and structural resemblance to the original.

3.1.2 Leakage Decision Module

This module defines the detection score function $\mathcal{S}_M(\cdot)$ and the indicator function $\mathbb{I}$ of SrDetection under both gray-box and black-box settings. The specific computation and decision rules are described in the following sections.

Gray-box Detection via Perplexity. Under the gray-box setting, we have access to the token-level probabilities of the target Code LLM M. For a given set $\mathcal{X} = \{x_0\} \cup \mathcal{V}$, consisting of the original sample x_0 and its semantic variants $\mathcal{V}$, the key observation is that, if the model has memorized x_0, its perplexity will be lower than that of any variant.

Accordingly, we define the detection score function $\mathcal{S}_M(x)$ as:

$$\mathcal{S}_M(x) \coloneqq \text{PPL}(x), \tag{2}$$

where the perplexity of sample x is $\text{PPL}(x) = \exp\big(-\frac{1}{|x|}\sum_{t=1}^{|x|} \log P_M(x_t \mid x_{<t})\big)$, and x_t denotes the t-th token in the sequence.

The final leakage decision under the gray-box setting is made using the indicator function $\mathbb{I}[\cdot]$:

$$\text{Leakage}_{\text{gray}}(x_0) = \mathbb{I}\Big[\mathcal{S}_M(x_0) < \min_{x_i \in \mathcal{V}} \mathcal{S}_M(x_i)\Big],$$

where $\mathbb{I}[\cdot] = 1$ if the condition holds (indicating leakage) and 0 otherwise.

By comparing scores only within the self-generated set $\mathcal{X}$, this approach remains threshold-free and directly implements the self-referential decision principle of SrDetection.

Black-box Detection via Generation Similarity. In the black-box setting, only the generated outputs of Code LLMs are accessible. For each sample $x_i \in \mathcal{X} = \{x_0\} \cup \mathcal{V}$, we construct a prefix x_i^{pre} and feed it to the model M to obtain a generated suffix $\hat{c}_i = M(x_i^{\text{pre}})$. We denote the ground-truth suffix as c_i^{gt}.

Then, the detection score function under the black-box setting is defined as the n-gram overlap (NGO) between the generated and ground-truth suffix:

$$\mathcal{S}_M(x_i) \coloneqq \text{NGO}(\hat{c}_i, c_i^{\text{gt}}), \tag{3}$$

where the NGO of sample x is $\text{NGO}(\hat{c}_i, c_i^{\text{gt}}) = \frac{|\text{N-grams}(\hat{c}_i) \cap \text{N-grams}(c_i^{\text{gt}})|}{|\text{N-grams}(c_i^{\text{gt}})|}$, and $\text{N-grams}(\cdot)$ denotes the set of all consecutive N-token sequences in

a code snippet. This metric robustly captures the surface-level similarity of code sequences.

The final leakage decision under black-box setting is made using the indicator function $\mathbb{I}[\cdot]$:

$$\text{Leakage}_{\text{black}}(x_0) = \mathbb{I}\Big[\mathcal{S}_M(x_0) > \max_{x_i \in \mathcal{V}} \mathcal{S}_M(x_i)\Big].$$

Here, $\mathbb{I}[\cdot]$ outputs 1 if the condition holds (indicating leakage) and 0 otherwise.

Overall, by comparing detection scores only within the self-generated set $\mathcal{X}$, this approach remains threshold-free and directly implements the self-referential decision principle of SrDetection.

3.2 Leakage Testbed Construction

To evaluate code leakage detection methods under precise conditions, we constructed a general testbed capable of supporting different datasets and open-source Code LLMs.

Dataset and Model Preparation. We first selected a publicly available code benchmark dataset to serve as the evaluation corpus. To minimize pre-existing exposure, we filtered samples by computing the PPL of each original sample and its generated variants under the target model, removing any sample whose original form was already the easiest (lowest PPL) among its variants. The remaining samples were then split equally into training and test subsets to support controlled evaluation.

Simulating Data Leakage via Continued Pre-Training. To emulate realistic code memorization, each model was further pre-trained on the training subset while keeping the test subset held out. To reflect typical pre-training pipelines, where code represents only a small fraction of the overall corpus, the training subset was mixed with a publicly available, well-known pre-training corpus. Detection performance was then evaluated on the held-out test subset under this mixed-corpus training setup.

4 Experimental Settings

In this section, we present the evaluation metrics and a set of competitive baseline methods. Detailed implementation settings and dataset statistics are reported in Appendix A.

Evaluation Metrics. We evaluated detection performance using standard binary classification metrics: Accuracy, Precision, Recall, and F1-score.

Baseline Methods. We compared SrDetection against competitive baselines in both gray-box and black-box settings: **(1) Gray-box Baselines.** These methods required access to token-level probabilities: **PPL** (Gonen et al., 2023) used raw perplexity as the detection signal, where lower perplexity indicates a higher likelihood of memorization; **Zlib** (Carlini et al., 2021) computed the ratio between model perplexity and Zlib compression entropy to normalize for text complexity; **Lowercase** (Carlini et al., 2021) computed the perplexity ratio between the original text and its lowercase variant; **Min-K% Prob** (Shi et al., 2024) used the average log probability of the lowest $K\%$ tokens as the detection score, assuming non-member samples contain outlier tokens; **Min-K%++** (Zhang et al., 2025) extended Min-K% Prob with local-mode normalization for improved robustness; **DC-PDD** (Zhang et al., 2024) incorporated divergence calibration between the model distribution and token frequency statistics. **(2) Black-box Baselines.** These methods operated using only model-generated text: **VeilProbe** (Hu et al., 2025) learned a sequence-to-sequence mapping between inputs and model outputs, using token perturbations to strengthen detection; **DPDLLM** (Zhou et al., 2024) used a reference model to extract features from generated text for membership classification.

5 Results and Analysis

5.1 Main Results

The performance of SrDetection against state-of-the-art gray-box and black-box baselines across different continued pre-training epochs is reported in Table 1 and Table 2.

Gray-box Setting. As shown in Table 1, SrDetection consistently outperformed all gray-box baselines across both model families and all reported epochs. At 1 epoch, it achieved F1 scores of 74.76 on Qwen2.5-7B and 69.99 on LLaMA3.1-8B, substantially higher than competing approaches, many of which remained random predictions. Some baselines (e.g., Min-K%++ and Lowercase) showed gains at specific epochs, but performance was unstable and model-dependent. In contrast, SrDetection remained strong and stable across epochs and models. This robustness arises from comparing each sample to its semantics-preserving variants, which provides an internal reference and avoids reliance on externally calibrated thresholds.

Black-box Setting. Table 2 reports black-box performance. On Qwen2.5-7B, SrDetection surpassed all baselines at every epoch, with F1 improving from 53.09 at 1 epoch to 82.06 at 5 epochs. On

Table 1: Detection performance comparison (F1-Macro, %) between SrDetection and gray-box baselines on the continued pre-trained Qwen2.5-7B and Llama3.1-8B models across different training epochs. The **bold** number indicates the best performance, while the underlined number represents the second-best.

Epochs	Methods	Qwen2.5-7B				Llama3.1-8B			
		Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
1	PPL (Gonen et al., 2023)	50.85	71.94	50.62	34.88	50.34	64.03	50.11	33.77
	Lowercase (Carlini et al., 2021)	59.09	69.93	58.92	52.44	66.95	69.16	66.88	65.91
	Zlib (Carlini et al., 2021)	50.24	25.12	50.00	33.44	50.25	58.45	50.02	33.52
	Min-K% Prob (Shi et al., 2024)	50.43	66.83	50.19	33.94	50.57	68.06	50.34	34.31
	Min-K%++ Prob (Zhang et al., 2025)	54.31	64.03	54.11	44.36	55.94	69.67	55.75	46.34
	DC-PDD (Zhang et al., 2024)	51.47	70.75	51.24	36.36	53.54	69.80	53.33	41.18
	SrDetection (Ours)	**74.95**	**75.00**	**75.81**	**74.76**	**70.01**	**70.10**	**70.03**	**69.99**
3	PPL (Gonen et al., 2023)	50.38	75.15	50.14	33.75	50.39	75.15	50.16	33.79
	Lowercase (Carlini et al., 2021)	52.27	73.51	52.05	37.98	53.94	75.78	53.73	41.26
	Zlib (Carlini et al., 2021)	50.28	75.13	50.05	33.54	50.29	75.13	50.07	33.59
	Min-K% Prob (Shi et al., 2024)	50.90	75.29	50.67	34.90	50.93	75.29	50.71	34.98
	Min-K%++ Prob (Zhang et al., 2025)	70.83	80.62	70.70	68.22	55.21	76.21	55.01	43.72
	DC-PDD (Zhang et al., 2024)	54.88	75.84	54.66	43.10	52.40	74.67	52.19	38.18
	SrDetection (Ours)	**88.59**	**89.70**	**88.55**	**88.50**	**82.43**	**86.42**	**82.36**	**81.92**
5	PPL (Gonen et al., 2023)	50.43	75.17	50.19	33.86	50.66	75.22	50.43	34.39
	Lowercase (Carlini et al., 2021)	59.80	77.06	59.61	51.97	57.91	77.06	57.72	48.63
	Zlib (Carlini et al., 2021)	50.28	75.13	50.05	33.54	50.27	75.12	50.05	33.54
	Min-K% Prob (Shi et al., 2024)	51.37	75.41	51.14	35.93	52.20	75.06	51.98	37.72
	Min-K%++ Prob (Zhang et al., 2025)	78.55	84.36	78.45	77.56	60.29	77.49	60.11	52.73
	DC-PDD (Zhang et al., 2024)	54.69	75.27	54.47	42.79	54.56	75.49	54.35	42.52
	SrDetection (Ours)	**90.53**	**91.56**	**90.49**	**90.47**	**84.47**	**87.96**	**84.41**	**84.09**

LLaMA3.1-8B, performance at 1 epoch was lower (F1 49.58) and comparable to VeilProbe. Nevertheless, SrDetection separated clearly from baselines at 3 and 5 epochs, indicating that the self-referential, code-aware comparison remained effective even under output-only access. Most baselines stayed near chance across epochs, underscoring the difficulty of black-box leakage detection and the advantage of relative comparison.

Gray-box vs. Black-box. Overall, gray-box evaluation yielded higher absolute performance due to access to token probabilities, whereas black-box evaluation relied only on generated outputs. Importantly, both settings shared the same self-referential framework and semantics-preserving comparison, differing only in the evidence available to the detector. This consistency suggests that SrDetection's effectiveness is driven by within-sample relative comparison rather than by specific access assumptions.

5.2 Sensitivity Analysis

In the following, we examine the sensitivity of SrDetection under two key factors: (1) the number of semantics-preserving variants generated per original sample, and (2) the proportion of code in the continued pre-training corpus. Since gray-box evaluation provides more direct access to model logits and generally yields more stable performance, we focus our detailed analysis on the gray-box setting. Black-box sensitivity results, which rely solely on generated outputs, are reported in Appendix B.

Variants Quantity. As shown in Table 3, detection performance improved as the number of semantics-preserving variants increased, supporting the benefit of relative comparison between each original sample and its variants. Even with a small number of variants (e.g., $n = 3$), SrDetection exceeded random baselines. Performance continued to improve over the tested range, suggesting that additional variants strengthen robustness across models and training epochs.

Mixing Ratio. Figure 3 reports results under different code-to-text mixing ratios, obtained by mixing the training subset with a well-known, publicly available pre-training dataset (Gao et al., 2020). We evaluated ratios ranging from 1:1, 1:5, to 1:10 (code to general web data). Across all ratios, SrDetection consistently outperformed baseline methods. While probability-based baselines were sensitive to the data mix ratio, the self-referential, threshold-free design of SrDetection remained stable, demon-

Table 2: Detection performance comparison (F1-Macro, %) between our method and black-box baselines on the continued pre-trained Qwen2.5-7B and Llama3.1-8B models across different training epochs. The **bold** number indicates the best performance, while the underlined number represents the second-best.

Epochs	Methods	Qwen2.5-7B				LLaMA3.1-8B			
		Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
1	DPDLLM (Zhou et al., 2024)	49.62	49.42	11.97	41.22	50.23	50.23	**100.00**	33.59
	VeilProbe (Hu et al., 2025)	51.04	51.10	50.96	49.33	51.56	51.60	51.58	**51.48**
	SrDetection (Ours)	**54.64**	**54.73**	**55.51**	**53.02**	**51.59**	**52.00**	51.68	49.58
3	DPDLLM (Zhou et al., 2024)	50.28	50.26	**98.68**	34.84	50.34	50.28	**99.95**	33.88
	VeilProbe (Hu et al., 2025)	50.95	50.93	50.91	50.68	51.27	51.28	51.28	51.25
	SrDetection (Ours)	**66.24**	**66.28**	66.78	**66.00**	**63.06**	**63.45**	63.09	**62.82**
5	DPDLLM (Zhou et al., 2024)	46.21	47.72	73.89	41.70	49.07	49.64	**95.35**	34.99
	VeilProbe (Hu et al., 2025)	51.75	51.83	51.69	50.78	51.61	51.64	51.63	51.53
	SrDetection (Ours)	**81.91**	**81.89**	**82.27**	**81.85**	**78.60**	**79.02**	78.58	**78.51**

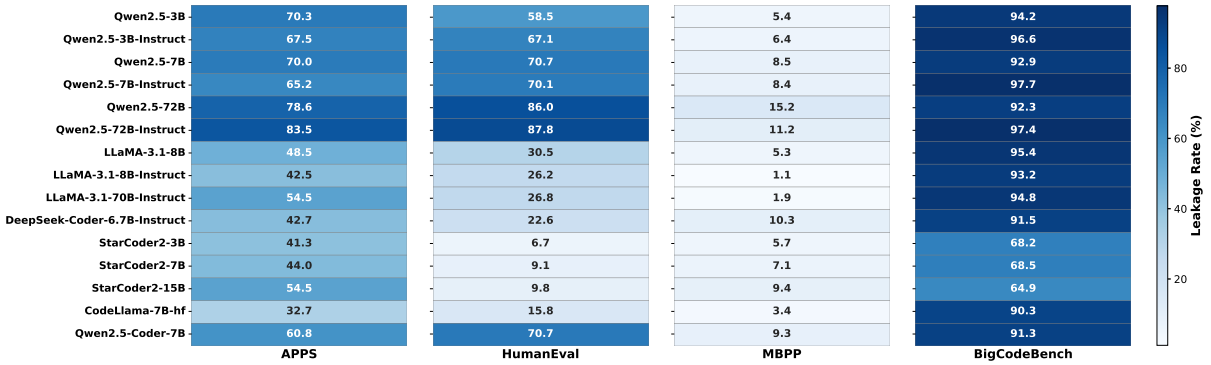

Figure 2: Data leakage rates (%) of mainstream Code LLMs on standard benchmarks in the gray-box setting.

Table 3: Detection performance (F1-Macro, %) under different numbers of variants and training epochs under the gray-box setting.

Model	Variants (n)	Training Epochs		
		1	3	5
Llama-3.1-8B	1	33.43	33.43	33.43
	3	65.30	69.65	72.03
	5	67.20	74.89	77.05
	10	**69.99**	**81.92**	**84.09**
Qwen2.5-7B	1	33.44	33.44	33.44
	3	69.32	75.56	77.45
	5	71.56	81.44	83.05
	10	**74.76**	**88.50**	**90.47**

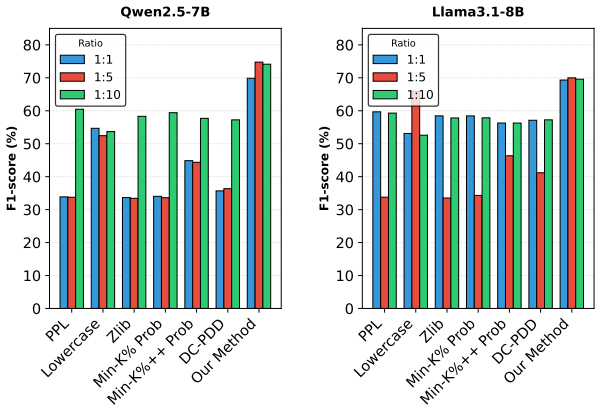

Figure 3: Detection performance (F1-score, %) under different mixed ratios during continued pre-training with 1 epoch in gray-box setting.

strating robustness to variations in pre-training corpus composition.

5.3 Data Leakage across Benchmarks

To assess leakage in widely used Code LLMs, we applied our gray-box detector to 15 publicly released models across four popular code generation benchmarks: APPS (Hendrycks et al., 2021), HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and BigCodeBench (Zhuo et al., 2025), as summarized in Figure 2. Leakage was evaluated directly from model behavior under gray-box access.

APPS exhibited consistently higher leakage for several Qwen2.5 variants, in line with overlap-based trends reported in prior work (Zhou et al., 2025), which used MinHash+LSH near-duplicate

Table 4: Thresholds and scores of baselines on sample #1061 under the gray-box setting. All baseline methods failed to detect leakage because their scores did not fall below the respective thresholds, whereas our method correctly identified the leaked sample.

Method	Score	Threshold	Correct?
PPL	-1.9485	-21.6516	✗
Lowercase	-0.9486	-0.9820	✗
Zlib	-0.0008	-0.0195	✗
Min-K%	-2.7820	-7.9785	✗
Min-K%++	-0.6970	-1.5204	✗
DC-PDD	0.0098	0.0090	✗
Our Method	–	–	✓

```python
def shipWithinDays(weights, D)
    :
  left = max(weights)
  right = left * len(weights)
     // D
  while left < right:
    mid = left + (right - left
     ) // 2
    c = 0
    d = 1
    for w in weights:
      if c + w <= mid:
        c += w
      else:
        d += 1
        c = w
    if d > D:
      left = mid + 1
    else:
      right = mid
  return left
# ...
```

(a) PPL=1.71

```python
def getMinimumDaysForShipping(
     cargo_weights, num_days)
     :
  left_limit = max(
     cargo_weights)
  right_limit = left_limit *
     len(cargo_weights) //
     num_days
  while left_limit <
     right_limit:
    mid_range = left_limit + (
     right_limit - left_limit
     ) // 2
    weight_sum = 0
    load_count = 1
    for cargo in cargo_weights
     :
      if weight_sum + cargo <=
       mid_range:
        weight_sum += cargo
      else:
# ...
```

(b) PPL=2.31

Figure 4: Comparison of an original code sample (a) and one of its semantics-preserving variants (b).

detection followed by manual verification. HumanEval showed moderate leakage, consistent with previous analyses indicating small but nonzero contamination. MBPP was largely unaffected for most models, reflecting its low leakage in the same study. Notably, BigCodeBench demonstrated high leakage in our gray-box evaluation, suggesting that behavior-based detection can reveal potential leakage beyond strict data overlap.

Overall, these results largely align with overlap-based findings while providing complementary, benchmark-specific insights. Related black-box experiments are reported in Appendix B.

5.4 Case Study

To illustrate why the self-referential design of the proposed SrDetection method succeeds where conventional detectors fail, we analyze some cases and present a representative code sample. As shown in Table 4, established gray-box baselines, including PPL, Zlib, and Min-K%, failed to detect leakage for this example. These methods typically rely on absolute scores and fixed thresholds; here, all scores remained above their thresholds, producing a false negative and underscoring the brittleness of thresholded, natural-language-oriented detectors on code.

In contrast, SrDetection evaluates each sample *relative* to its semantics-preserving variants. For the sample in Figure 4, we generated ten variants that preserved core logic while altering surface form (e.g., variable and function names). The original sample yielded the minimum perplexity (PPL=1.7082), while all variants had higher perplexity (2.0287–2.7565). This pronounced local minimum indicates memorization tied to the original surface form. As shown in Figure 4, which compares the original sample with one variant, SrDetection correctly flagged leakage while threshold-based baselines failed.

6 Discussion and Conclusion

SrDetection reframes data leakage detection for Code LLMs as a *self-referential* problem: instead of depending on inaccessible pretraining corpora, fragile temporal rules, or externally calibrated thresholds, it measures leakage through *relative* behavioral discrepancies between an original code sample and its semantics-preserving variants. This perspective is particularly well matched to code benchmarks, where surface forms can be altered without changing functionality, and where reuse and refactoring routinely break timestamp- and overlap-based assumptions. To support rigorous evaluation, we further introduce a controlled testbed that enables precise attribution of leakage by explicitly separating seen and unseen samples through continued pre-training. This testbed is model- and dataset-agnostic, and allows leakage detectors to be compared under reproducible conditions. Empirically, our results show that SrDetection can reliably identify leaked samples in both gray-box and black-box settings and can surface benchmark-specific contamination patterns that are not revealed by overlap checks alone. Beyond detection performance, these findings suggest that leakage is not a marginal artifact but a systematic confounder in current evaluation practice, reinforcing the need for leakage-aware benchmark design and reporting.

Limitations

While SrDetection demonstrates robust performance in detecting memorization of code samples, there remain opportunities for further improvement. The effectiveness of detection depends on the diversity and quality of semantics-preserving variants; exploring additional transformation strategies could strengthen the contrastive signal. In black-box settings, subtle output differences may be masked in very large or instruction-tuned models, suggesting that alternative or complementary signals could further enhance detection. Future work may also extend SrDetection to cover broader code modalities and to integrate adaptive transformation strategies that better reflect real-world code variability.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions to improve this paper. We used AI assistants only for minor language polishing and grammar correction. All research ideas, methodology, experiments, and conclusions were developed and verified by the authors. Min Yang was supported by National Key Research and Development Program of China (2024YFF0908200), National Natural Science Foundation of China (Grant No. 62376262), the Natural Science Foundation of Guangdong Province of China (2024A1515030166, 2025B1515020032). Shiwen Ni was supported by GuangDong Basic and Applied Basic Research Foundation (2023A1515110718 and 2024A1515012003) and Shenzhen Science and Technology Program (JCYJ20250604182917023).

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2633–2650. USENIX Association.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2024. Investigating data contamination in modern benchmarks for large language models. In *NAACL 2024*, pages 8706–8719. Association for Computational Linguistics.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12039–12050. Association for Computational Linguistics.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, and 1 others. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Hila Gonen, Srini Iyer, Terra Blevins, Noah A. Smith, and Luke Zettlemoyer. 2023. Demystifying prompts in language models via perplexity estimation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 10136–10148. Association for Computational Linguistics.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8):220:1–220:79.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Ruihan Hu, Yu-Ming Shang, Jiankun Peng, Wei Luo, Yazhe Wang, and Xi Zhang. 2025. Automated detection of pre-training text in black-box llms. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, IJCAI '25.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. 2024. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. *arXiv e-prints*, arXiv:2410.03981.

José Antonio Hernández López, Boqi Chen, Mootez Saad, Tushar Sharma, and Dániel Varró. 2025. On inter-dataset code duplication and data leakage in large language models. *IEEE Trans. Software Eng.*, 51(1):192–205.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On leakage of code generation evaluation datasets. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13215–13223. Association for Computational Linguistics.

Meta AI. 2024. Introducing meta llama 3: The most capable openly available llm to date.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024*, pages 14116–14137. Association for Computational Linguistics.

Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, Yann Ollivier, and Hervé Jégou. 2019. White-box vs black-box: Bayes optimal strategies for membership inference. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5558–5567. PMLR.

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2024. Detecting pretraining data from large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 3–18. IEEE Computer Society.

Congzheng Song and Vitaly Shmatikov. 2019. Auditing data provenance in text-generation models. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 196–206. ACM.

Llama Team. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

OpenAI Team. 2025. Chatgpt: optimizing language models for dialogue. 2022. *Accessed on*, 31.

Ruijie Xu, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. 2024. Benchmarking benchmark leakage in large language models. *CoRR*, abs/2404.18824.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024a. Qwen2.5 technical report. *CoRR*, abs/2412.15115.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. 2023. Rethinking benchmark and contamination for language models with rephrased samples. *CoRR*, abs/2311.04850.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2024b. Unveiling memorization in code models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 72:1–72:13. ACM.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 37:1–37:12. ACM.

Jingyang Zhang, Jingwei Sun, Eric Yeats, Yang Ouyang, Martin Kuo, Jianyi Zhang, Hao Frank Yang, and Hai Li. 2025. Min-k%++: Improved baseline for pre-training data detection from large language models. In *The Thirteenth International Conference on Learning Representations*.

Weichao Zhang, Ruqing Zhang, Jiafeng Guo, Maarten de Rijke, Yixing Fan, and Xueqi Cheng. 2024. Pretraining data detection for large language models: A divergence-based calibration method. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 5263–5274. Association for Computational Linguistics.

Baohang Zhou, Zezhong Wang, Lingzhi Wang, Hongru Wang, Ying Zhang, Kehui Song, Xuhui Sui, and Kam-Fai Wong. 2024. DPDLLM: A black-box framework for detecting pre-training data from large language

models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 644–653, Bangkok, Thailand. Association for Computational Linguistics.

Xin Zhou, Martin Weyssow, Ratnadira Widyasari, Ting Zhang, Junda He, Yunbo Lyu, Jianming Chang, Beiqi Zhang, Dan Huang, and David Lo. 2025. Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks. *CoRR*, abs/2502.06215.

Terry Yue Zhuo, Minh Chien Vu, 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, and 2 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

A Implementation Details

A.1 SrDetection

Table 5: Statistics of the filtered APPS-based datasets after PPL-based exclusion.

Model	Number of Instances	
	Training	Test
Qwen2.5-7B	1061	1051
LLaMA-3.1-8B	2216	2196

Table 6: Threshold values used by baseline methods across training epochs (1, 3, 5) for Qwen2.5-7B and LLaMA3.1-8B, organized by method and epoch.

Method	Qwen2.5-7B	LLaMA3.1-8B
1 Epoch		
PPL	-21.6516	-23.4101
Lowercase	-0.9820	-0.9681
Zlib	-0.0195	-0.0188
Min-K% Prob	-7.9785	-7.9442
Min-K%++ Prob	-1.5204	-1.5833
DC-PDD	0.0090	0.0091
3 Epochs		
PPL	-36.1829	-36.1829
Lowercase	-1.0078	-1.0078
Zlib	-0.0217	-0.0217
Min-K% Prob	-9.5799	-9.5799
Min-K%++ Prob	-5.8413	-5.8413
DC-PDD	0.0082	0.0082
5 Epochs		
PPL	-86.3298	-86.3298
Lowercase	-1.0021	-1.0021
Zlib	-0.0272	-0.0272
Min-K% Prob	-11.8104	-11.8104
Min-K%++ Prob	-18.4400	-18.4400
DC-PDD	0.0073	0.0073

All experiments are implemented using PyTorch[2] and the Hugging Face Transformers library[3], running on a system with four NVIDIA A100 GPUs. We evaluate our proposed method, SrDetection, under both gray-box and black-box scenarios. For SrDetection, we generate $n = 10$ semantic-preserving variants per code sample to enable self-referential comparison. In the black-box setting, we additionally employ N-gram overlap (NGO) with $N = 7$. To create semantic variants, we design prompt templates targeting two key transformations: identifier renaming (Table 7) and test case replacement (Table 8). These templates preserve the core logic, functionality, and overall code structure of the original samples while introducing sufficient syntactic diversity to challenge conventional detectors. Table 6 provides a supplementary reference by reporting the threshold values used by the baseline methods across different models with different training epochs (1, 3, 5), corresponding to the performance results shown in Table 1 in the main text.

Table 7: Prompt template for identifier renaming.

User:
You are given a Python code snippet:
{{python_code_snippet}}

Task:
Generate 10 alternative versions of the snippet by renaming function names, parameters, and variable identifiers.
Constraints:
– Preserve comments, formatting, indentation, spacing, and line breaks exactly.
– Update identifiers consistently if they appear in comments.
– Do not change the logic or functionality.

Return each version as a single string, using \n for newlines.

Table 8: Prompt template for test case transformation.

User:
You are given a programming problem:
{{question}}

Provided Test Cases:
Input: *{{test_case_input}}*
Output: *{{test_case_output}}*

Task:
Replace the sample test cases in the problem with the provided examples.
Constraints:
– Preserve structure, formatting, indentation, and line breaks.
– Only replace input/output sections.
– Do not modify any other text.

Return the modified problem as a single string, using \n for newlines.

A.2 Leakage Testbed

In our study, we instantiated the leakage testbed using the APPS dataset (Hendrycks et al., 2021) and selected representative open-source Code LLMs as target models, including Qwen2.5-7B (Yang et al., 2024a) and LLaMA-3.1-8B (Team, 2024). Table 5 reports the number of instances in the filtered APPS-based datasets after PPL-based exclusion. To support controlled evaluation, the dataset was split into training and test subsets at a 1:1 ratio, resulting in model-specific dataset sizes summarized in Table 5. To simulate realistic pre-training

[2] https://pytorch.org
[3] https://huggingface.co/docs/transformers

conditions in which code constitutes a minor portion of the overall corpus, the APPS training subset was mixed with general web data from RefinedWeb (Gao et al., 2020). A mixing ratio of 1:5 (code to general web data) was adopted to approximate typical code concentrations. All models undergo continued pre-training on our filtered datasets using LoRA (Hu et al., 2022) with a learning rate of 1.0×10^{-4}, batch size of 32, and training epochs of 1, 3, and 5.

B Supplementary Black-box Experiments

In addition to the gray-box analyses reported in the main text, we conducted a series of experiments under the black-box setting.

Black-box Detection Across Models and Benchmarks. Table 9 presents data leakage detection rates (%) for several widely used Code LLMs across three benchmarks: HumanEval, MBPP, and BigCodeBench. Our method achieves consistently higher detection rates compared with baseline approaches, highlighting its ability to identify potential leakage even with limited access to model internals. Notably, leakage is more pronounced on BigCodeBench for most models, consistent with trends observed in gray-box evaluations.

Sensitivity to Data Composition. Table 10 reports F1-Macro performance under varying code-to-text mixing ratios during continued pre-training (1 epoch). SrDetection maintains strong, stable performance across all ratios (1:1, 1:5, 1:10), while baseline methods exhibit more variable results, demonstrating the resilience of our self-referential, threshold-free approach under different data distributions.

Impact of Variant Quantity. Table 11 shows performance under varying numbers of semantic-preserving variants. Increasing the number of variants generally improves detection, confirming that relative comparisons between the original sample and its variants are effective even when only the generated outputs are available. This trend mirrors observations in the gray-box setting, indicating that the self-referential mechanism is robust to the level of access provided.

Overall, these black-box experiments complement the gray-box analyses, demonstrating that SrDetection's relative, variant-based framework can reliably detect code leakage even under restricted observational settings.

Table 9: Data leakage detection rates (%) across different Code LLMs and benchmarks (black-box).

Model	Humaneval	MBPP	BigCodeBench
Qwen2.5-7B	16.46	10.68	52.94
LLaMA-3.1-8B	7.93	9.45	61.19
DeepSeek-Coder-6.7B-Instruct	7.93	10.78	35.21
StarCoder2-7B	7.93	6.16	35.03
CodeLlama-7B-hf	11.59	12.73	49.96
Qwen2.5-Coder-7B	18.90	18.17	44.78

Table 10: Detection performance (F1-Macro, %) under different mixed ratios during continued pre-training with 1 epoch under the black-box setting.

Mixed Ratio	Method	Qwen2.5-7B	Llama3.1-8B
1:1	PPL	33.86	59.68
	Lowercase	54.68	53.11
	Zlib	33.65	58.45
	Min-K% Prob	34.03	58.44
	Min-K%++ Prob	44.85	56.28
	DC-PDD	35.69	57.12
	SrDetection (Ours)	**69.86**	**69.33**
1:5	PPL	33.75	33.77
	Lowercase	52.44	65.91
	Zlib	33.44	33.52
	Min-K% Prob	33.61	34.31
	Min-K%++ Prob	44.36	46.34
	DC-PDD	36.36	41.18
	SrDetection (Ours)	**74.76**	**69.99**
1:10	PPL	60.46	59.29
	Lowercase	53.68	52.58
	Zlib	58.32	57.82
	Min-K% Prob	59.41	57.86
	Min-K%++ Prob	57.70	56.27
	DC-PDD	57.24	57.24
	SrDetection (Ours)	**74.13**	**69.58**

Table 11: Detection performance (F1-Macro, %) under different numbers of variants and training epochs under the black-box setting.

Model	Variants (n)	Training Epochs		
		1	3	5
Qwen2.5-7B	1	52.57	58.35	65.17
	3	55.15	63.39	75.67
	5	54.61	64.86	78.67
	10	53.02	66.00	81.85
Llama-3.1-8B	1	50.96	55.91	64.45
	3	51.95	60.05	72.38
	5	51.48	61.74	74.74
	10	49.58	62.82	78.51