The Poisoned Chalice

An Exploratory Study Into Polarized Augment Calibration for Membership Inference in Code LLMs

Roham Koohestani[1]

Supervisor(s): Dr. Maliheh Izadi[1]**, Ir. Jonathan B. Katzy**[1]**, Ir. Ali Al-Kaswan**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 25, 2026

Name of the student: Roham Koohestani
Final project course: CSE3000 Research Project
Thesis committee: Dr. Maliheh Izadi, Ir. Jonathan B. Katzy, Ir. Ali Al-Kaswan, Prof.dr.ir. Inald Lagendijk

Abstract

Large Language Models (LLMs) for code are trained on large amounts of data that may contain copyrighted and licensed content, which motivates internal auditing methods that can test whether specific data points were included during training. In this work we conduct an exploratory evaluation of membership inference attacks (MIAs) as auditing signals for code-specialized LLMs. We compare a loss-based baseline to Polarized Augment Calibration (PAC) across three open models in the 3B–4B range (Mellum-4B, StarCoder2-3B, and SmolLM3-3B) using the Java subset of a contamination-controlled evaluation dataset. We find that PAC provides consistent improvements over the loss signal on the code models, while near-members samples are detected almost as effectively as exact members. A stratified analysis shows that attack performance varies substantially with file properties, with strongest separability on small-to-medium files and on code with higher alphanumeric content, and degradation on very large files. Motivated by the syntactic fragility of token-swap augmentation on code, we propose PAC-AST, an AST-guided augmentation scheme that generates syntactically valid neighbors. PAC-AST exhibits improved behavior on larger and syntactically complex files where token-swap PAC degrades but underperforms in smaller and alphanumeric-rich strata due in part to a reduced effective mutation magnitude. Overall, the results indicate that (i) calibration-based signals can strengthen grey-box auditing for code models, (ii) dataset and program characteristics are major drivers of measured leakage, and (iii) code-specific augmentation is a promising direction but requires controlling perturbation magnitude and neighbor quality to yield stable gains.

1 Introduction

Large Language Models (LLMs) are trained on large quantities of data [3]. This practice raises data governance concerns, as these datasets are frequently compiled with minimal oversight and contain large quantities of personal, private, and copyrighted content [11]. This lack of governance has led to several legal challenges [19; 20]. Recent litigation has raised claims of copyright infringement involving the use of proprietary text for model training [19].

The central question often revolves around the matter of *fair use* in which courts (predominantly in the United States) have issued rulings that vary between cases [20]. With no clear legal precedent, the burden has shifted toward requiring concrete technical proof of data misuse. Plaintiffs have used techniques intended to demonstrate verbatim memorization of their copyrighted text. One such technique is called Training Data Extraction (TDE) [4]. Although TDE is an established approach, it remains an attack that requires identification of exact memorized sequences [26]. A more scalable alternative to TDE is a Membership Inference Attack (MIA) [22]. Rather than extracting data, an MIA aims to solve a binary classification problem, asking: Given a data point, was it part of the model's training set. MIAs are widely used as a proxy to measure privacy leakage and the degree of model memorization [11].

The matter of memorization is also a concern in specialized domains such as software engineering. Previous work has demonstrated that code LLMs are vulnerable to TDE attacks and that they tend to memorize content such as *license information* at a higher rate than other code structures [1]. This vulnerability introduces legal risks of copyright and license infringement [11]. This work explicitly identifies and discusses MIAs as a methodology for detecting the inclusion of copyright-protected or licensed content in model training data.

To mitigate these risks, organizations require auditing mechanisms to verify if specific licensed content was consumed during training. While external adversaries typically rely on black-box access, internal auditing permits a *white-box* setting which assumes full access to model parameters and gradients. However, despite the necessity for reliable internal auditing, the field lacks a clear consensus on which signals are most effective for large-scale models.

Recent surveys highlight a lack of systematic surveys that address their effectiveness and limitations [23]. While *Loss Attack* [22] provides a computationally efficient baseline, newer methods like Polarized Augment Calibration (PAC) [24] claim superior performance through more involved calibrations. The comparative reliability of these simple versus complex signals remains untested in the domain of source code.

This research is structured as an exploratory study in which the findings of each stage inform the methodology of the next. We begin by establishing a baseline comparison of attack performance (RQ1). Based on these results, we investigate how specific data characteristics influence attack success (RQ2). Finally, identifying limitations in the standard augmentation approach for code, we propose and evaluate a code-specific adaptation (RQ3).

This leads to the following concrete sub-questions:

1. **RQ1 (Performance):** How does the attack performance of PAC compare to the loss attack on code-specific LLMs?
2. **RQ2 (Robustness):** How do various characteristics of data affect the effectiveness of each method?
3. **RQ3 (Code-Specific Alternative):** How does the performance of a code-specific variant of PAC compare?

Summary of findings. Across the evaluated 3B–4B models, PAC yields higher separability than the loss-based baseline on the code-specialized models (StarCoder2 and Mellum), while improvements are smaller on the general-purpose control (SmolLM3). Near-duplicate samples are detected at rates comparable to exact members, indicating that minor edits do not reliably remove membership signal under these scores. A stratified analysis shows that performance varies substantially with file characteristics. We find that separability is strongest on small-to-medium files and on code with higher alphanumeric content, and it degrades on very large and syntactically complex files. Motivated by this large-file degradation, we introduce PAC-AST, which replaces token swaps with AST-guided perturbations; PAC-AST is more stable in large/complex strata where token-swap PAC degrades, but it underperforms on small and alphanumeric-rich code, consistent with its smaller effective mutation magnitude in those cases.

In the sections to follow, we begin by presenting an overview of the background literature along with a positioning of our work in the related field section 2. We then proceed to present our exploratory analysis, organized by research question. Each subsequent section details the specific method, setup, and results for that inquiry. Finally, we provide a discussion of our results section 6 along with the concluding remarks section 7. We make all code and logs of the runs publicly available for reproducibility purposes. All the results and code used in this study along with additional tables and material are provided as part of the replication package of this study [13].

2 Background and Related Work

In this section, we situate our work within the broader context of existing research and discuss key developments that have shaped the field.

2.1 Membership Inference Attacks

We begin by outlining a taxonomy to clarify what we mean by the different categories of attacks.

Taxonomy. A Membership Inference Attack (MIA) is defined as a binary classification problem [22; 8]. The attack's premise is that models exhibit distinguishable statistical outputs (e.g., higher confidence or lower perplexity) for data encountered during training compared to unseen data [22]. The goal of the attacker is to build a classifier that can detect this behavioral difference.

The effectiveness and design of an MIA are dictated by the attacker's level of access to the target model. This access is formally categorized into a three-tier taxonomy [23; 8]:

- **Black-Box:** The adversary has no internal access and can only provide an input and observe the final output of the model (e.g., the generated text or a label) [22].
- **Grey-Box:** The adversary has partial access, such as the ability to see the model's output probabilities or confidence scores for its predictions, but not its internal parameters or gradients [23].
- **White-Box:** The adversary has complete access to the model, including its architecture, all parameters (weights), and the ability to compute internal states like loss and gradients [15; 6; 17].

  This project focuses on the grey-box setting where the auditor can compute token-level log-probabilities (logits) for the model.

Attack Signals for Grey-Box Inference. Within the grey-box setting, the primary distinction between methods lies in the signal they use to classify membership.

The most foundational method is the **Loss Attack**, which uses the model's loss as the primary signal [25]. The core assumption is that a model's loss will be observably lower for a member sample it has already seen than for a non-member sample. In a black-box setting, this loss must be inferred using computationally expensive shadow models [22]. In our grey-box setting, the exact loss can be computed directly via a single forward pass. This makes it computationally efficient compared to methods requiring shadow model training. However, many studies argue that a single scalar loss value provides insufficient information and can be seen as a weak signal, as it fails to capture the model's complex internal state [17].

More complex methods use richer signals. The **Polarized Augment Calibration (PAC)** attack is a reference-free method that claims to be a plug-and-play solution for grey-box settings [24]. It uses a two-part mechanism:

1. **A Polarized Distance Signal:** Instead of average loss, it computes a *polarized distance* (L_M) that compares the log-probabilities of the most likely (MAX-k_1%) tokens to the least likely (MIN-k_2%) tokens. This builds on concepts from other "reference-free" attacks like Min-K% Prob [21]. In this work, we refer to tokens in (MAX-k_1%) as k_{near} and to those in (MIN-k_2%) as k_{far}.
2. **Augment Calibration:** To avoid shadow models, PAC calibrates this L_M score by generating *adjacent samples* ($\tilde{z}$) via *randomly swapping pairs of N tokens* within the original input. The attack signal is the difference between the original sample's score and the average score of its augmented non-member neighbors [24].

The core assumption of this augmentation, that a random token-swap creates a valid adjacent sample, is a primary, untested weakness of the PAC method when it comes to syntactically rigid data like source code or natural language. This matter will be discussed thoroughly in the results (RQ3) and discussion section of this thesis.

2.2 Data Governance and Auditing in Code

The need for robust MIA-based auditing is particularly relevant in the domain of software engineering. Code LLMs, including the target models for this study (e.g., StarCoder2, Mellum, and SmolLM), are trained on massive (often unsanitized) datasets scraped from the internet [14; 2; 18]. The StarCoder2 model, for instance, was trained on "The Stack v2," a corpus derived from over 600 programming languages [14].

This large-scale data ingestion creates a severe data governance problem, as these datasets inevitably contain code with restrictive "copyleft" licenses (e.g., GPL, AGPL) [11]. As previously stated, previous work has established that code LLMs are both vulnerable to TDE attacks [1], and preferentially memorize legally-sensitive content, including **license information**, at a higher rate than other code structures [1]. The verbatim memorization of this code by a model can place its users in a state of license infringement [11]. This motivates the need for reliable auditing tools to detect the inclusion of such copyrighted and licensed content.

To address governance challenges, the industry is moving toward formal standardization, most notably the **ISO/IEC 42001:2023** [10] standard for Artificial Intelligence Management Systems (AIMS). This framework mandates auditable processes for the entire AI lifecycle. The IBM Granite family recently became the first open-source model series to achieve this certification by requiring concrete proofs of data provenance and license verification [9]. This rise in formal compliance standards further amplifies the need for technical auditing tools, such as Membership Inference Attacks, that can independently verify these claims.

This highlights a gap in the existing literature. While the practical need for reliable, well-understood auditing tools is clear, the comparative strengths and weaknesses of these different grey-box attack signals are poorly understood. Recent studies highlight an absence of comprehensive evaluations that systematically examine their strengths and weaknesses in large-scale models [23], and that substantial knowledge gaps remain regarding the impact of different signals on attack efficacy. To our knowledge, no work has directly compared the robustness and failure modes of the PAC method's augmentation strategy against the loss attack in the domain of source code. This exploratory study aims to fill that precise gap.

3 RQ1: Comparative Performance of Attacks

We begin by establishing a baseline for attack performance. The goal of this section is to determine how the Polarized Augment Calibration (PAC) method compares to the standard grey-box loss attack when applied to code-specialized LLMs.

3.1 Method: Attack Definitions

We compare two methods on code LLMs trained for completion. Given a sample z (source code snippet) and a trained model f_θ, the goal of an MIA is to decide whether z was part of the training set of f_θ (member) or not (non-member). We treat near-members samples as a separate category of interest, evaluated as a member.

Loss-based. The baseline commonly used for grey-box testing is the Loss Attack [25; 8], which unlike earlier models that train shadow models directly uses the model's loss to infer membership. Concretely. For a given model f_θ and a tokenised sequence $x = (x_1, \ldots, x_T)$, we compute the standard autoregressive training loss

$$\ell(x) \;=\; -\frac{1}{T-1}\sum_{t=1}^{T-1} \log p_\theta(x_{t+1} \mid x_{\le t}).$$

The loss is computed after truncating the sequence to a maximum length L (by default $L = 8192$ tokens). The membership score for

the loss attack is defined as $s_{\text{loss}}(x) = -\ell(x)$, so that higher scores correspond to lower loss and thus a higher likelihood of membership.

PAC-based attack. PAC extends the MIA framework by defining a *polarized distance* over token log-probabilities and calibrating it using augmented neighbors of the input [24].

Polarized distance: Given a sequence of per-token log-probabilities p, sorted into $p_{(1)} \leq \cdots \leq p_{(T)}$, the polarized distance is defined as:

$$L_M(p; k_{\text{near}}, k_{\text{far}}) \;=\; \frac{1}{k_{\text{far}}} \sum_{i=0}^{k_{\text{far}}-1} p_{(T-i)} \;-\; \frac{1}{k_{\text{near}}} \sum_{i=1}^{k_{\text{near}}} p_{(i)}.$$

L_M measures the gap between the most confidently predicted tokens and the least confidently predicted tokens. In this context, *confidence* refers to the conditional probability assigned by the model to the actual token observed in the sequence ($p_\theta(x_t \mid x_{<t})$). A high probability implies the model effectively anticipates the token based on learned patterns, while a low probability indicates a "surprising" or unexpected continuation.

Augmentation-based calibration: PAC calibrates the polarized distance by contrasting the original sample with a small set of perturbed *adjacent* samples. For RQ1, we adopt the standard mechanism: generating n adjacent samples by randomly swapping a fraction m of the tokens (default $m = 0.3$). The PAC score is defined as:

$$s_{\text{pac}}(x) \;=\; L_M(x) \;-\; \frac{1}{n} \sum_{i=1}^{n} L_M(\tilde{x}_i).$$

3.2 Experimental Setup

Source data. Our experiments are based on the Java subset of The Heap [12], a multilingual, contamination-free code dataset designed for evaluating code LLMs. From this split we obtain three conceptual groups of files: *members*, *near-members* (near-members variants of the member files), and *non-members*. Each group initially contains exactly 50 000 Java files. We then perform a stratified sampling to the dataset (as described further in section 4) to obtain a representative sample from the dataset. We publish the resulting sampled dataset to the HuggingFace Hub.[1] Throughout the thesis, "member" and "non-member" refer to membership under the dataset construction used for evaluation. This does not fully determine whether a sample was present in a model's complete pretraining mixture when that mixture is not publicly specified. Additionally, the result presentations we make use of the following abbreviations: (1) **Members-All (MA)** refers to the union of the set of member and near-member samples. (2) **Member-Exact (ME):** refers to the set of exclusively exact members, and (3) **Member-Near (MN)** refers to near-members, [2] and, finally (4) **Non-Members (NM)** refers to datapoints not included in the training dataset.

Models. We evaluate the attacks on three open-weight LLMs in the 3B–4B parameter range:

- **Mellum-4b-base:** Designed for in-IDE code completion [18].
- **StarCoder2-3B:** Trained on The Stack v2 and code-related sources [14].
- **SmolLM3-3B:** A general language model, which we use as a control model in our experiments [2].

[1] https://huggingface.co/datasets/RohamKoohestani/ResearchProjectSampled

[2] The reader is refereed to the original dataset paper for the exact process through which near-members are detected and flagged [12]

Evaluation protocol. We report ROC-AUC and PR-AUC as threshold-independent measures of separability. ROC-AUC summarizes ranking quality across operating points, while PR-AUC is sensitive to false positives and is therefore informative when there is a potential class imbalance (potentially applicable in our models under evaluation). For completeness, we also report accuracy, precision, recall, and F1 at a single operating point, using the threshold obtained by using Youden's J. All experiments are executed on DelftBlue, the TU Delft supercomputer using NVIDIA V100 GPUs [7]. Additionally, we use the same default parameters as in the original PAC attack paper, specifically $k_{near} = 30$, $k_{far} = 5$, a mutation ratio of $m = 0.3$, and a neighborhood size of $n = 5$ adjacent samples.

3.3 Results: Aggregate Performance

Table 1 presents the aggregate performance metrics for both attacks. The loss-based attack provides a consistent baseline across all code-specific models. For StarCoder2 and Mellum, the attack achieves an ROC-AUC of approximately 0.63 on the aggregate (MA vs. NM) task. Interestingly, the general-purpose model, SmolLM3, exhibits lower vulnerability to the loss attack (ROC-AUC 0.573) compared to the dedicated code models.

PAC surpasses the loss-based attack on the StarCoder2 model. For the aggregate task (MA vs. NM), PAC achieves an ROC-AUC of 0.662 compared to the loss attack's 0.625. This performance gap is consistent across Exact Members and Near Members. The PR-AUC also shows an increase from 0.750 (Loss) to 0.798 (PAC). On the general-purpose SmolLM3 model, the performance gap between PAC and Loss is less pronounced.

An observation across all experiments is the high detection rate of Near-Members (MN). For StarCoder2 under the PAC attack, the ROC-AUC for Near-Members (0.661) is very close to that of Exact Members (0.662). This indicates that the attack is robust to minor modifications.

Summary of RQ1. The loss attack provides a solid baseline, with code models showing higher vulnerability than SmolLM3. PAC improves performance on StarCoder2 and Mellum, and maintains comparable results on SmolLM3. In all cases, the attacks detect Near-Members almost as effectively as Exact Members.

4 RQ2: Impact of Data Characteristics

Having established the aggregate performance in RQ1, we observe variance in attack success across models. To understand the drivers of this variance, we conduct an exploratory analysis into how specific characteristics of the source code affect the effectiveness of each method.

4.1 Method: Stratified Analysis

We apply a stratified sampling procedure to the dataset to analyze performance across diverse dimensions. Instead of sampling files uniformly at random, we construct a stratified sample to maintain diversity. For each file, we compute:

- **Code size**, based on the number of characters or tokens;
- **Alphanumeric fraction,** defined as the fraction of characters in the set {A–Z, a–z, 0–9} among all characters in the file.
- **Syntactic size**, i.e., the number of AST nodes, reflecting the number of distinct syntactic constructs;
- **Type**, indicating member, near-member, or non-member.

Table 1: Attack performance comparison. The best **ROC-AUC** and **PR-AUC** scores per model are marked in **bold**. The scenario yielding the highest ROC-AUC for each model is underlined.

Attack	Model	Scenario	ROC-AUC	PR-AUC	Acc.	Prec.	Rec.	F1
loss [22]	Mellum	MA vs. NM	0.628	0.764	0.565	0.762	0.497	0.602
		ME vs. NM	0.629	0.628	0.598	0.627	0.480	0.544
		MN vs. NM	0.626	0.617	0.601	0.611	0.499	0.549
	SmolLM	MA vs. NM	0.573	0.701	0.595	0.705	0.666	0.685
		ME vs. NM	0.572	0.539	0.561	0.557	0.585	0.571
		MN vs. NM	0.575	0.541	0.560	0.540	0.668	0.597
	StarCoder2	MA vs. NM	0.625	0.750	0.580	0.750	0.546	0.632
		ME vs. NM	0.622	0.600	0.596	0.601	0.562	0.581
		MN vs. NM	0.629	0.605	0.600	0.604	0.519	0.558
pac [24]	Mellum	MA vs. NM	0.696	**0.827**	0.613	0.807	0.545	0.651
		ME vs. NM	**0.712**	0.742	0.662	0.730	0.511	0.601
		MN vs. NM	0.679	0.691	0.634	0.660	0.516	0.579
	SmolLM	MA vs. NM	0.578	**0.735**	0.517	0.738	0.418	0.533
		ME vs. NM	0.579	0.591	0.566	0.588	0.431	0.498
		MN vs. NM	**0.578**	0.579	0.568	0.581	0.414	0.483
	StarCoder2	MA vs. NM	**0.662**	**0.798**	0.585	0.791	0.505	0.617
		ME vs. NM	**0.662**	0.680	0.624	0.654	0.522	0.581
		MN vs. NM	0.661	0.664	0.625	0.649	0.504	0.567

Table 2: Bucket definitions used in the stratified analysis, with sample counts (n) for the MA vs. NM evaluation subset. File size is measured in characters (pre-tokenization) and syntactic size is the AST node count.

Dimension	Bucket	Range	n
File size (chars)	Very Small	[1, 1074]	2225
	Small	(1074, 1946]	2607
	Medium	(1946, 3369]	2875
	Large	(3369, 6844]	2694
	Very Large	(6844, 1419061]	2083
Alphanumeric fraction	Mid-Low	[0.2, 0.4]	363
	Mid	(0.4, 0.6]	4191
	Mid-High	(0.6, 0.8]	5119
	High	(0.8, 1]	2806
Syntactic size (nodes)	Simple	[1, 58]	4137
	Moderate	(58, 211]	4987
	Complex	(211, 51010]	3360

Each unique combination of bucket assignments defines a stratum. We then draw a sample from each stratum proportional to the number of files it contains. One thing to note is that we treat AST node count as a syntactic-size proxy (construct count), which is known to be strongly correlated with logical lines of code in prior work [16]. We present bounds for each of the buckets in Table 2.

4.2 Experimental Setup

For this analysis, we utilize the same models and scoring mechanisms as RQ1, but we aggregate results per stratum. We restrict our reporting in this section to the aggregate All Members (MA) vs. Non-Members (NM) scenario to provide a holistic view as we noticed minimal variability in the results from RQ1.

4.3 Results: Robustness and Strata

The results for this section have been visualized as plots in Figure 1. We provide the detailed performance for both attacks in the appendix.

Impact of File Size We observe a non-linear relationship between code size and attack performance, which somewhat resembles an inverted U-shaped curve. Across all models, attack performance peaks at the *Small* and *Medium* file sizes and degrades significantly for *Very Large* files. For the loss attack on StarCoder2, the ROC-AUC drops from a peak of 0.689 (Small) to 0.520 (Very Large). PAC demonstrates better robustness to this degradation. While PAC also experiences a drop, it maintains an ROC-AUC of 0.630 on very large files for the same model.

Impact of AST Node Count We observe a negative association between AST node count and membership inference performance in our stratified analysis. For both attacks, the *Simple* bucket yields higher detection rates than the *Complex* bucket. For instance, on the Mellum model using the PAC attack, the ROC-AUC decreases from 0.697 (Simple) to 0.672 (Complex).

Impact of Alphanumeric Ratio The alphanumeric ratio acts as a proxy for the density of natural language (e.g., variable names, comments) versus raw logic and literals. We observe that code with a *High* alphanumeric ratio is significantly easier to detect. On StarCoder2, the PAC attack achieves an ROC-AUC of 0.718 for the High stratum, compared to 0.642 for the Mid-High stratum. This suggests that LLMs show signs of membership for natural language patterns and unique identifiers more readily than abstract logic.

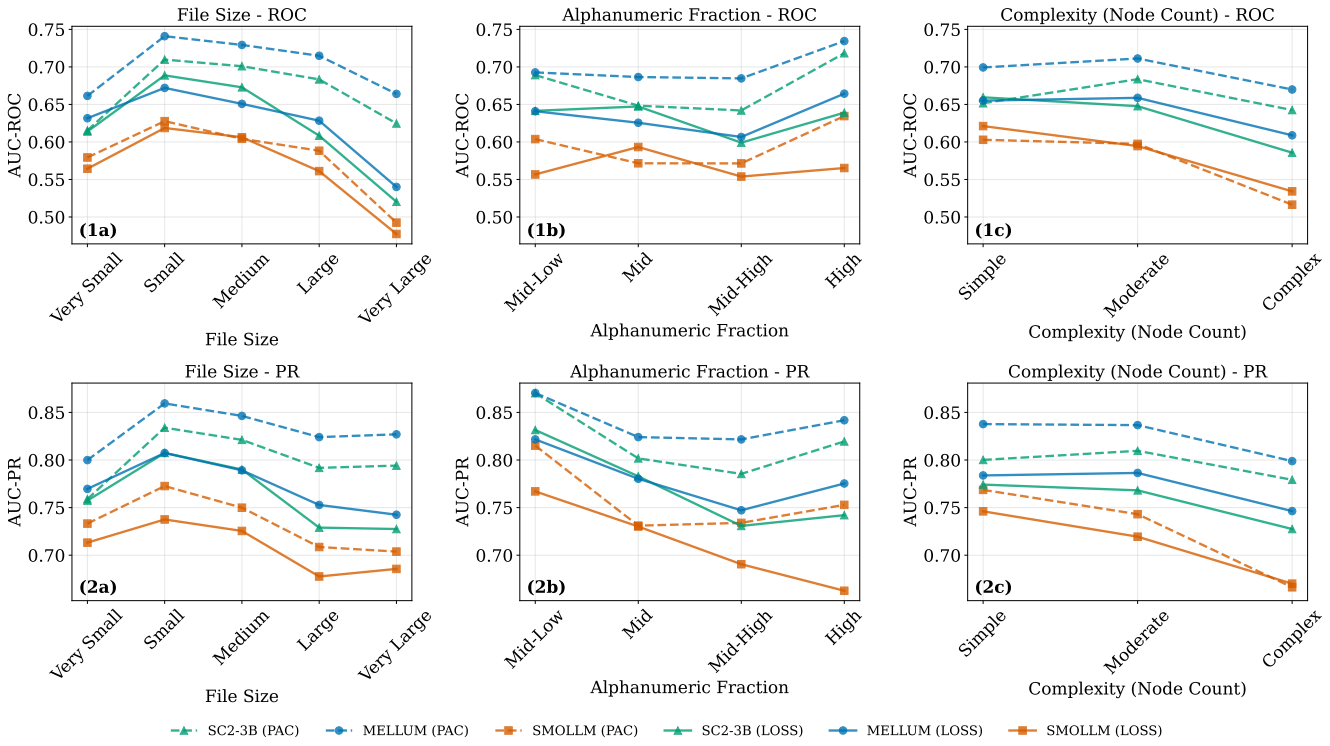

Figure 1: Summary of experimental results for models and attacks grouped into categories based on different stratification of code attributes. Results for ROC-AUC (top row), and PR-AUC (bottom row).

Summary of RQ2. Attack performance is highly dependent on data characteristics. **Small-to-Medium sized files** with **High Alphanumeric Ratios** (rich in identifiers/comments) and **Low Complexity** represent the most vulnerable strata. While both attacks degrade on Very Large files, PAC is significantly more robust than the Loss baseline in these edge cases.

5 RQ3: Code-Specific Adaptation

Our analysis in RQ1 and RQ2 suggests that while PAC is robust in certain easier cases, the underlying assumption of its augmentation strategy that random token swaps create valid "adjacent" samples may be ill-suited for the rigid syntax of source code. Token-level swaps generally break syntactic correctness, potentially introducing confounding factors. In this final exploratory step, we propose and evaluate a code-specific adaptation.

5.1 Method: AST-Based Augmentation

To address the limitation of token swaps, we introduce a code-specific adaptation of the augmentation procedure. Rather than perturbing the token sequence directly, we operate over the abstract syntax tree (AST) of the input program. An abstract syntax tree (AST) is a hierarchical tree-structured representation of the syntactic structure of source code in a programming language, where each node corresponds to a language construct and extraneous syntactic details (such as punctuation and grouping symbols) are omitted to focus on essential structure [5]. The central idea in our approach is to generate adjacent non-members by swapping structurally comparable AST nodes. This preserves core syntactic and lexical properties while producing controlled perturbations.

Given a program x, we parse it into an AST and identify categories of nodes that are safe to permute, such as swapping local variable declarations, literal values, or statement-level constructs with others of the same grammatical form. The perturbed AST $\tilde{a}$ is then converted back to source code $\tilde{x}$ using a standard AST-to-text unparser. The code-specific PAC score is defined as:

$$s_{\text{pac-ast}}(x) = L_M(x) - \frac{1}{n}\sum_{i=1}^{n} L_M(\tilde{x}_i),$$

where each $\tilde{x}_i$ is derived from an AST-guided perturbation. We present the pseudocode for this approach in algorithm 1.

5.2 Experimental Setup

We use the same experimental framework as previous sections. For the AST adaptation, we parse the Java files using a Java parser.[3] Files that cannot be parsed are removed. We compare the performance of $s_{\text{pac-ast}}$ against the standard s_{pac} (token swap) and s_{loss}.

5.3 Results

Effective perturbation magnitude differs between PAC and PAC-AST. Although both methods use the same nominal mutation ratio ($m = 0.3$), the *effective* amount of change in the resulting neighbor samples differs substantially. Token-swap PAC maintains a change ratio close to 0.35 across syntactic-size buckets. In contrast, PAC-AST yields substantially smaller changes for syntactically small files, with the median change ratio increasing from approximately 0.05 (Simple) to 0.08 (Moderate) and 0.17 (Complex) (Figure 5). This indicates that, under the current node-swap strategy,

[3] We use the tree-sitter library for this purpose

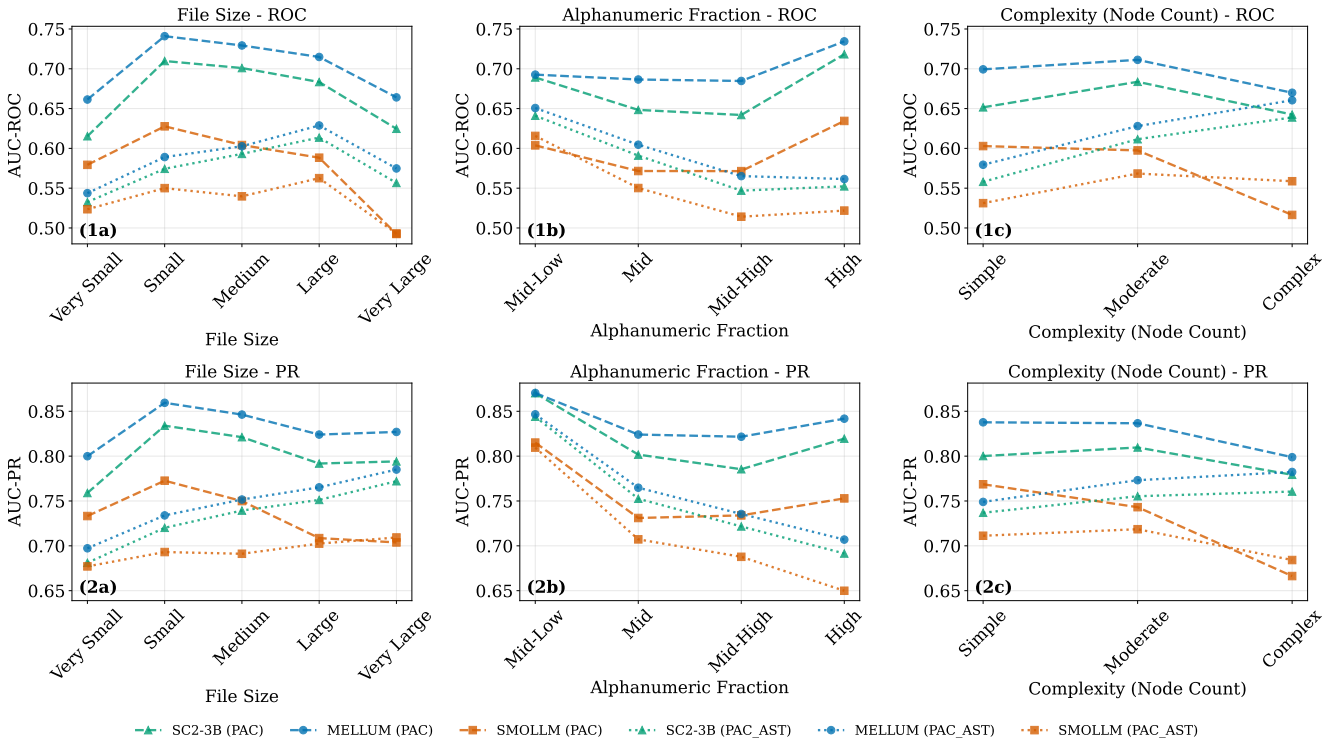

Figure 2: PAC vs. PAC-AST stratified by file size, alphanumeric fraction, and syntactic size (node count) for MA vs. NM.

syntactically small programs receive fewer or smaller perturbations than intended by the nominal m parameter.

PAC-AST improves with file size and syntactic size, while token-swap PAC degrades. Across MA vs. NM, ME vs. NM, and MN vs. NM, we observe that token-swap PAC exhibits a peak in Small-to-Medium buckets and degrades on Very Large files (Figure 2). PAC-AST shows a different trend, namely, for the code-specialized models (StarCoder2 and Mellum), both ROC-AUC and PR-AUC tend to increase from Very Small to Large, and remain more stable in the Large-to-Very Large regime than token-swap PAC. A similar pattern holds when stratifying by syntactic size (node count): PAC-AST improves from Simple to Complex buckets on StarCoder2 and Mellum, whereas token-swap PAC decreases or remains flat.

PAC-AST degrades with higher alphanumeric fraction. When stratifying by alphanumeric fraction, token-swap PAC improves in the High bucket, consistent with RQ2. In contrast, PAC-AST shows a monotone decrease in both ROC-AUC and PR-AUC as alphanumeric fraction increases (Figure 2). This indicates that the current AST-guided perturbations do not preserve, or do not sufficiently perturb, the same lexical cues that are predictive under token swaps in text-heavy code (identifiers and comments).

Score moments indicate increasing dynamic range for PAC-AST on larger syntactic sizes. We observe opposite trends in the mean and variance of the calibrated PAC score across syntactic-size buckets. For token-swap PAC, both the mean and variance decrease from Simple to Complex. For PAC-AST, both moments increase from Simple to Complex across all models. Combined with the performance trends above, this suggests that AST-guided perturbations yield scores with larger spread on syntactically larger files, which is consistent with increased separability in those buckets.

Summary of RQ3. PAC-AST ***does not*** dominate token-swap PAC across all strata. Its current implementation under-mutates syntactically small files, which coincides with lower performance in Small and Simple buckets. However, PAC-AST improves with file size and syntactic size on code models, where token-swap PAC degrades. PAC-AST also shows an opposing trend on alphanumeric-rich code, where token-swap PAC remains stronger.

6 Discussion

In this work, we set out to evaluate the efficacy of grey-box Membership Inference Attacks (MIAs) as auditing tools for code-specialized Large Language Models. Our results highlight distinct behaviors in how code models show signals of membership training data compared to general-purpose models, and how specific data characteristics, such as file size and the density of natural language, influence this vulnerability.

Vulnerability of Specialized Code Models Our baseline analysis (RQ1) revealed that specialized code models (StarCoder2 and Mellum) exhibit a higher susceptibility to membership inference compared to the general-purpose control model (SmolLM3). While the Loss Attack provided a consistent baseline, the PAC method demonstrated superior performance on the StarCoder2 model. This discrepancy suggests that the focused training procedures of code LLMs may inadvertently lower the threshold for memorization compared to models trained on broader, more diverse internet text. This aligns with previous findings indicating that code models tend to memorize specific high-frequency structures, such as license text, at higher rates than other content [1]. Consequently, organizations deploying specialized code models face a heightened audit risk, as these models retain stronger signals of their training data.

Input: Code sample x, mutation ratio m, max tries T
Output: Perturbed Sample $\tilde{x}$

GENERATENEIGHBOR(x, m, T) **for** $t \leftarrow 1$ **to** T **do**
  $\mathcal{T} \leftarrow$ PARSE(x);
  $\mathcal{C} \leftarrow$ COLLECTNODESBYCATEGORY$(\mathcal{T})$;
  $\mathcal{C}' \leftarrow \{c \in \mathcal{C} \mid |c| \geq 2\}$;
  if $\mathcal{C}' = \emptyset$ **then**
    return x
  end
  $N \leftarrow \sum_{c \in \mathcal{C}'} |c|$;
  $B \leftarrow \max(1, \lfloor m \cdot N/2 \rfloor)$;
  $\{B_c\}_{c \in \mathcal{C}'} \leftarrow$ ALLOCATEBUDGETS$(B, \{|c|\})$;
  $x' \leftarrow x$;
  foreach $c \in \mathcal{C}'$ **do**
    Re-parse x' to refresh node offsets;
    Select $2B_c$ nodes from c without replacement;
    $x' \leftarrow$ SWAPNODES$(x'$, selected nodes$)$;
  end
  if $x' \neq x$ **then**
    return x'
  end
end

Algorithm 1: Code-Specific PAC Attack (PAC-AST)

Membership inference for near-members An interesting finding from our experiments is the model's ability to detect Near-Members with an accuracy virtually identical to that of Exact Members. For instance, on StarCoder2, the PAC attack achieved an ROC-AUC of 0.663 for Near-Members, indistinguishable from the 0.664 achieved for exact members. This has significant implications for data governance and copyright enforcement. It suggests that minor code refactoring is insufficient to remove the provenance of the data from the model's memory. The model appears to memorize the underlying semantic structure or unique syntactic patterns rather than just the verbatim token sequence. For auditors, this validates the use of MIAs as robust tools for detecting license infringement, even when the source code in the training set has undergone slight modifications.

The "Sweet Spot" of Memorization Our stratified analysis (RQ2) delineates a clear "sweet spot" for attack success. We observed that files with a High Alphanumeric Ratio (rich in natural language, comments, and identifier names) are significantly easier to detect than those with low ratios. This indicates that LLMs may latch onto the unique entropy of natural language embedded within code (e.g., specific comment strings or unique variable naming conventions) more readily than abstract logic or control flow structures. Furthermore, we identified a non-linear relationship regarding file size and complexity. Attack performance peaks at Small to Medium file sizes and degrades for Very Large files. Similarly, we found that fewer AST nodes correlates with higher detection rates. This suggests that "boilerplate" code or smaller, well-commented utility functions are more vulnerable to leakage than massive, complex monolithic files. This degradation on large files may stem from the dilution of the loss signal over long sequences, a limitation the PAC method partially mitigates through its calibration mechanism, maintaining higher robustness in these edge cases compared to the Loss baseline.

Augmentation quality and scaling effects in code. RQ3 evaluates whether syntax-preserving neighbors improve PAC calibration on source code. The results show that the effect depends on program size and code characteristics. PAC-AST improves as files become larger and syntactically more complex, while token-swap PAC degrades in the Large and Very Large buckets. This pattern is consistent across MA/ME/MN discrimination settings. A factor that co-varies with this trend is the effective perturbation magnitude, namely in that token-swap PAC maintains a near-constant change ratio, whereas PAC-AST yields a much smaller change ratio on syntactically small files and increases with node count. This implies that the current comparison confounds augmentation strategy (AST-guided vs. token swap) with perturbation magnitude.

Alphanumeric-rich code remains a hard case for the current PAC-AST design. PAC-AST degrades as alphanumeric fraction increases, opposing the trend of token-swap PAC. Since AST-based perturbations typically exclude in-comment (textual) perturbations and operate on a limited subset of syntactic categories, the generated neighbors may not perturb identifier- and text-level cues at the same rate as token swaps. This indicates that a code-specific calibration strategy may require hybrid neighbors that combine syntactic validity with targeted lexical perturbations (e.g., identifier renaming within scope, literal perturbations, or comment-level transformations when present).

Implications for improving PAC-AST. The results motivate an adaptive mutation budget for AST-guided perturbations. One approach is to target an effective change ratio rather than a fixed node mutation ratio by iteratively sampling additional swaps until a token-level change budget is reached, or by scaling the number of swaps inversely with syntactic size in small files. Another approach is to expand the set of swappable constructs (for example, more statement categories and scoped renaming) to increase the availability of valid perturbations in syntactically small programs.

6.1 Threats to validity

Metric overlap. AST node count is a syntactic-size proxy and is known to correlate strongly with lines-of-code style measures. As a result, our stratification dimensions (file size and AST node count) are not independent, and some effects attributed to one dimension may partially reflect the other.

Non-equivalent perturbation magnitude. PAC and PAC-AST share the same nominal m value, but the effective change ratio differs substantially across syntactic-size buckets. Since calibration depends on the properties of the neighbor distribution, performance differences may be partly attributable to perturbation magnitude rather than the augmentation mechanism itself. A controlled comparison would match PAC and PAC-AST on an effective change budget (for example, a fixed token-level change ratio).

Parsing and reconstruction effects. PAC-AST requires parsing and reconstructing source code. Files that fail to parse are excluded, which may bias the sample toward syntactically regular code. In addition, reconstruction may modify formatting, whitespace, or minor lexical details, which can affect tokenization and per-token probabilities even when the AST structure is preserved.

Residual contamination and label noise. Our evaluation defines membership relative to the dataset construction and our experimental splits. For models trained on broad mixtures of code and text, membership in the model's pretraining data is not fully observable. As a result, some samples labeled as non-members under our dataset definition may still have been observed during pretraining. This risk is larger for models whose pretraining corpora extend beyond the sources used to define membership in our evaluation setting (e.g., SmolLM3). Such label noise can reduce measured separability and can distort comparisons across models if contamination rates differ between them. The results should therefore be interpreted as membership inference under the dataset membership definition rather than as definitive evidence of inclusion or exclusion from the complete pretraining mixture.

Near-member labeling ambiguity. We treat near-members samples as members, but high similarity does not imply that a near-member was observed during model training. In particular, small files may match due to common boilerplate, high-frequency language constructs, or reused license templates. This can overestimate robustness of membership signals under near-members transformations.

Sequence truncation and coverage. All scores are computed on sequences truncated to a maximum length (8192 tokens, dependent on the tokenizer of the model used). For large files this means that only a prefix contributes to the measured loss and token probabilities. This can affect comparisons across file-size buckets and can partially explain degradation on very large files. It also interacts with augmentation because perturbations are applied within the truncated region.

6.2 Responsible Research

We used AI-based assistance during this project in two contexts. First, we used writing support tools (Writeful and ChatGPT) to improve grammar, phrasing, and LaTeX formatting in the manuscript. Second, we used JetBrains AI Assistant during development of the experimental pipeline for code navigation and suggested refactorings. The author retains responsibility for the technical content, experimental design, and interpretation. All numerical results and plots were produced by the implemented pipeline and were reviewed for consistency against intermediate logs. We do not rely on AI-generated content as evidence. We aim to support reproducibility by releasing the code and run results, and we restrict reported artifacts to derived statistics rather than releasing any model training data. All claims in the thesis are grounded in the cited literature or in the reported experiments, independent of AI-assisted editing.

7 Conclusion

This thesis evaluated grey-box membership inference attacks (MIAs) as auditing signals for code-specialized LLMs. Across three 3B–4B models and a contamination-controlled Java evaluation setting, PAC consistently outperformed a loss-based baseline on the code models, and near-members were detected almost as effectively as exact members. A stratified analysis showed that attack performance varies substantially with program characteristics, with stronger separability on small-to-medium and alphanumeric-rich code and degradation on very large files.

Motivated by the syntactic fragility of token-swap augmentation, we proposed PAC-AST, which generates syntactically valid neighbors via AST-guided perturbations. PAC-AST improves with file size and syntactic size in settings where token-swap PAC degrades, but underperforms on syntactically small and alphanumeric-rich code, partly due to a smaller effective mutation magnitude in small programs. Overall, the results indicate that calibration-based signals can strengthen internal auditing for code LLMs, while code-specific augmentation is promising but requires controlling perturbation magnitude and neighbor quality. A key limitation is that "non-member" is defined relative to the evaluation dataset and may not exclude membership in a model's full pretraining mixture when that mixture is unknown.

References

[1] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[2] Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, Xuan-Son Nguyen, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM3: smol, multilingual, long-context reasoner. https://huggingface.co/blog/smollm3, 2025.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[5] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022.

[6] Daniel DeAlcala, Aythami Morales, Julian Fierrez, Gonzalo Mancera, and Ruben Tolosana. gmint: Gradiant-based membership inference test applied to image models. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 2781–2790, 2025.

[7] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[8] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S Yu, and Xuyun Zhang. Membership inference attacks on machine learning: A survey. *ACM Computing Surveys (CSUR)*, 54(11s):1–37, 2022.

[9] IBM. Ibm granite, Jul 2025. https://www.ibm.com/granite.

[10] ISO. Iso/iec 42001:2023, Dec 2023. https://www.iso.org/standard/42001.

[11] Jonathan Katzy, Razvan Popescu, Arie Van Deursen, and Maliheh Izadi. An exploratory investigation into code license infringements in large language model training datasets. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 74–85, 2024.

[12] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models, 2025.

[13] Roham Koohestani. Replication package —— bachelor's thesis roham koohestani —— the poisoned chalice: An exploratory study into polarized augment calibration for membership inference in code llms, January 2026. https://doi.org/10.5281/zenodo.18367988.

[14] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder

2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[15] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE symposium on security and privacy (SP)*, pages 739–753. IEEE, 2019.

[16] Huy Nguyen, Michelle Lim, Steven Moore, Eric Nyberg, Majd Sakr, and John Stamper. Exploring metrics for the analysis of code submissions in an introductory data science course. In *LAK21: 11th International Learning Analytics and Knowledge Conference*, pages 632–638, 2021.

[17] Yan Pang, Tianhao Wang, Xuhui Kang, Mengdi Huai, and Yang Zhang. White-box membership inference attacks against diffusion models. *Proc. Priv. Enhancing Technol.*, 2025.

[18] Nikita Pavlichenko, Iurii Nazarov, Ivan Dolgov, Ekaterina Garanina, Dmitry Ustalov, Ivan Bondyrev, Kseniia Lysaniuk, Evgeniia Vu, Kirill Chekmenev, Joseph Shtok, et al. Mellum: Production-grade in-ide contextual code completion with multi-file project understanding. *arXiv preprint arXiv:2510.05788*, 2025.

[19] Audrey Pope. Nyt v. openai: The times's about-face, April 2024.

[20] Ropes & Gray. A tale of three cases: How fair use is playing out in ai copyright lawsuits, July 2025.

[21] Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. Detecting pretraining data from large language models. In *NeurIPS 2023 Workshop on Regulatable ML*.

[22] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[23] Hengyu Wu and Yang Cao. Membership inference attacks on large-scale models: A survey. *arXiv preprint arXiv:2503.19338*, 2025.

[24] Wentao Ye, Jiaqi Hu, Liyao Li, Haobo Wang, Gang Chen, and Junbo Zhao. Data contamination calibration for black-box llms. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 10845–10861, 2024.

[25] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st computer security foundations symposium (CSF)*, pages 268–282. IEEE, 2018.

[26] Weichen Yu, Tianyu Pang, Qian Liu, Chao Du, Bingyi Kang, Yan Huang, Min Lin, and Shuicheng Yan. Bag of tricks for training data extraction from language models, 2023.

A Detailed Results for RQ2 & RQ3

A note on confidence intervals: For each model and bucket, we estimate 95% confidence intervals for ROC-AUC and PR-AUC using a nonparametric bootstrap with 1,000 resamples (sampling bucket examples with replacement); we report the 2.5th and 97.5th percentiles of the bootstrap distribution and discard resamples that contain only one class.

Table 3: Loss attack outcomes across data characteristics for all member dataset samples (both near and exact), compared to Non-Member samples. The values between the brackets indicate the confidence interval of the result.

attack	model	stratification	bucket	ROC-AUC	PR-AUC
loss	StarCoder2	Alphanumeric Ratio	High	0.639 [0.619, 0.658]	0.742 [0.717, 0.764]
			Mid	0.647 [0.629, 0.664]	0.783 [0.766, 0.799]
			Mid-High	0.599 [0.583, 0.616]	0.731 [0.713, 0.749]
			Mid-Low	0.641 [0.576, 0.706]	0.832 [0.777, 0.877]
		File Size	Large	0.608 [0.587, 0.629]	0.729 [0.704, 0.752]
			Medium	0.673 [0.652, 0.693]	0.790 [0.768, 0.809]
			Small	0.689 [0.669, 0.710]	0.807 [0.787, 0.827]
			Very Large	0.520 [0.493, 0.548]	0.727 [0.702, 0.755]
			Very Small	0.614 [0.589, 0.637]	0.757 [0.732, 0.781]
		Syntactic Size (Node Count)	Complex	0.586 [0.565, 0.607]	0.727 [0.707, 0.749]
			Moderate	0.648 [0.631, 0.662]	0.768 [0.752, 0.784]
			Simple	0.659 [0.642, 0.676]	0.774 [0.755, 0.793]
	Mellum	Alphanumeric Ratio	High	0.664 [0.645, 0.684]	0.775 [0.755, 0.795]
			Mid	0.626 [0.608, 0.642]	0.780 [0.764, 0.797]
			Mid-High	0.607 [0.589, 0.622]	0.747 [0.730, 0.764]
			Mid-Low	0.641 [0.573, 0.706]	0.822 [0.766, 0.874]
		File Size	Large	0.628 [0.608, 0.650]	0.753 [0.730, 0.776]
			Medium	0.651 [0.630, 0.673]	0.789 [0.770, 0.809]
			Small	0.672 [0.650, 0.691]	0.807 [0.787, 0.825]
			Very Large	0.540 [0.513, 0.567]	0.743 [0.715, 0.768]
			Very Small	0.632 [0.608, 0.655]	0.770 [0.745, 0.793]
		Syntactic Size (Node Count)	Complex	0.609 [0.591, 0.630]	0.746 [0.728, 0.767]
			Moderate	0.659 [0.644, 0.674]	0.786 [0.771, 0.801]
			Simple	0.655 [0.638, 0.673]	0.784 [0.766, 0.803]
	SmolLM	Alphanumeric Ratio	High	0.565 [0.545, 0.587]	0.663 [0.638, 0.687]
			Mid	0.593 [0.575, 0.612]	0.730 [0.712, 0.751]
			Mid-High	0.554 [0.538, 0.571]	0.691 [0.672, 0.710]
			Mid-Low	0.557 [0.486, 0.624]	0.767 [0.705, 0.822]
		File Size	Large	0.561 [0.541, 0.584]	0.678 [0.652, 0.702]
			Medium	0.606 [0.583, 0.628]	0.725 [0.701, 0.749]
			Small	0.619 [0.595, 0.639]	0.738 [0.714, 0.762]
			Very Large	0.477 [0.451, 0.504]	0.686 [0.659, 0.715]
			Very Small	0.564 [0.539, 0.589]	0.713 [0.685, 0.741]
		Syntactic Size (Node Count)	Complex	0.534 [0.514, 0.554]	0.670 [0.647, 0.692]
			Moderate	0.594 [0.578, 0.611]	0.719 [0.702, 0.738]
			Simple	0.621 [0.604, 0.639]	0.746 [0.726, 0.765]

Table 4: PAC attack outcomes across data characteristics for all member dataset samples (both near and exact), compared to Non-Member samples. The values between the brackets indicate the confidence interval of the result.

attack	model	stratification	bucket	ROC-AUC	PR-AUC
pac	StarCoder2	Alphanumeric Ratio	High	0.718 [0.700, 0.737]	0.820 [0.802, 0.836]
			Mid	0.648 [0.632, 0.664]	0.802 [0.788, 0.816]
			Mid-High	0.642 [0.626, 0.658]	0.785 [0.769, 0.802]
			Mid-Low	0.689 [0.627, 0.745]	0.870 [0.828, 0.906]
		File Size	Large	0.683 [0.662, 0.704]	0.792 [0.769, 0.811]
			Medium	0.701 [0.683, 0.721]	0.821 [0.803, 0.839]
			Small	0.710 [0.690, 0.729]	0.834 [0.816, 0.851]
			Very Large	0.625 [0.600, 0.650]	0.794 [0.769, 0.818]
			Very Small	0.615 [0.592, 0.637]	0.759 [0.736, 0.781]
		Syntactic Size (Node Count)	Complex	0.643 [0.624, 0.661]	0.779 [0.760, 0.797]
			Moderate	0.684 [0.669, 0.699]	0.810 [0.795, 0.825]
			Simple	0.652 [0.634, 0.669]	0.800 [0.783, 0.817]
	Mellum	Alphanumeric Ratio	High	0.734 [0.716, 0.753]	0.842 [0.825, 0.856]
			Mid	0.687 [0.670, 0.702]	0.824 [0.809, 0.837]
			Mid-High	0.685 [0.668, 0.699]	0.822 [0.808, 0.836]
			Mid-Low	0.693 [0.630, 0.752]	0.870 [0.832, 0.908]
		File Size	Large	0.715 [0.695, 0.735]	0.824 [0.805, 0.841]
			Medium	0.729 [0.711, 0.747]	0.846 [0.830, 0.862]
			Small	0.741 [0.720, 0.760]	0.859 [0.844, 0.874]
			Very Large	0.664 [0.639, 0.689]	0.827 [0.806, 0.847]
			Very Small	0.661 [0.639, 0.683]	0.800 [0.779, 0.820]
		Syntactic Size (Node Count)	Complex	0.670 [0.653, 0.689]	0.799 [0.782, 0.815]
			Moderate	0.711 [0.698, 0.726]	0.837 [0.824, 0.849]
			Simple	0.699 [0.683, 0.715]	0.838 [0.823, 0.852]
	SmolLM	Alphanumeric Ratio	High	0.634 [0.613, 0.656]	0.753 [0.733, 0.775]
			Mid	0.572 [0.553, 0.590]	0.731 [0.712, 0.750]
			Mid-High	0.571 [0.554, 0.587]	0.734 [0.716, 0.752]
			Mid-Low	0.604 [0.536, 0.671]	0.815 [0.763, 0.862]
		File Size	Large	0.588 [0.566, 0.611]	0.709 [0.684, 0.734]
			Medium	0.604 [0.583, 0.627]	0.750 [0.728, 0.772]
			Small	0.628 [0.605, 0.649]	0.773 [0.749, 0.795]
			Very Large	0.492 [0.465, 0.520]	0.704 [0.677, 0.731]
			Very Small	0.579 [0.555, 0.603]	0.733 [0.705, 0.758]
		Syntactic Size (Node Count)	Complex	0.516 [0.498, 0.537]	0.666 [0.645, 0.688]
			Moderate	0.597 [0.581, 0.613]	0.743 [0.726, 0.761]
			Simple	0.603 [0.585, 0.621]	0.769 [0.751, 0.785]

Table 5: PAC-AST attack outcomes across data characteristics for all member dataset samples (both near and exact), compared to Non-Member samples. The values between the brackets indicate the confidence interval of the result.

attack	model	stratification	bucket	ROC-AUC	PR-AUC
pac_ast	StarCoder2	Alphanumeric Ratio	High	0.552 [0.532, 0.574]	0.691 [0.670, 0.715]
			Mid	0.591 [0.573, 0.610]	0.753 [0.735, 0.771]
			Mid-High	0.547 [0.530, 0.564]	0.722 [0.704, 0.738]
			Mid-Low	0.641 [0.584, 0.703]	0.844 [0.800, 0.888]
		File Size	Large	0.614 [0.593, 0.635]	0.751 [0.729, 0.773]
			Medium	0.593 [0.572, 0.616]	0.739 [0.716, 0.761]
			Small	0.574 [0.551, 0.598]	0.720 [0.696, 0.745]
			Very Large	0.557 [0.532, 0.582]	0.772 [0.749, 0.794]
			Very Small	0.533 [0.507, 0.558]	0.681 [0.655, 0.711]
		Syntactic Size (Node Count)	Complex	0.639 [0.620, 0.659]	0.761 [0.743, 0.781]
			Moderate	0.612 [0.596, 0.627]	0.755 [0.739, 0.771]
			Simple	0.558 [0.540, 0.578]	0.737 [0.719, 0.755]
	Mellum	Alphanumeric Ratio	High	0.561 [0.539, 0.583]	0.707 [0.683, 0.729]
			Mid	0.605 [0.587, 0.622]	0.765 [0.747, 0.781]
			Mid-High	0.565 [0.548, 0.581]	0.736 [0.719, 0.752]
			Mid-Low	0.651 [0.583, 0.711]	0.847 [0.802, 0.889]
		File Size	Large	0.629 [0.610, 0.650]	0.765 [0.744, 0.787]
			Medium	0.603 [0.581, 0.624]	0.752 [0.730, 0.774]
			Small	0.589 [0.567, 0.613]	0.734 [0.710, 0.759]
			Very Large	0.575 [0.550, 0.599]	0.785 [0.763, 0.806]
			Very Small	0.544 [0.519, 0.567]	0.697 [0.671, 0.722]
		Syntactic Size (Node Count)	Complex	0.660 [0.642, 0.678]	0.782 [0.765, 0.799]
			Moderate	0.628 [0.611, 0.644]	0.773 [0.758, 0.788]
			Simple	0.579 [0.560, 0.597]	0.749 [0.730, 0.769]
	SmolLM	Alphanumeric Ratio	High	0.522 [0.502, 0.544]	0.650 [0.625, 0.677]
			Mid	0.550 [0.533, 0.570]	0.707 [0.687, 0.727]
			Mid-High	0.514 [0.498, 0.531]	0.688 [0.670, 0.707]
			Mid-Low	0.616 [0.548, 0.680]	0.810 [0.757, 0.867]
		File Size	Large	0.563 [0.541, 0.585]	0.702 [0.677, 0.727]
			Medium	0.540 [0.517, 0.563]	0.691 [0.668, 0.715]
			Small	0.550 [0.528, 0.572]	0.693 [0.669, 0.717]
			Very Large	0.493 [0.466, 0.521]	0.709 [0.682, 0.737]
			Very Small	0.524 [0.498, 0.549]	0.677 [0.649, 0.706]
		Syntactic Size (Node Count)	Complex	0.559 [0.539, 0.580]	0.684 [0.662, 0.707]
			Moderate	0.568 [0.553, 0.586]	0.719 [0.701, 0.737]
			Simple	0.531 [0.513, 0.549]	0.711 [0.692, 0.729]

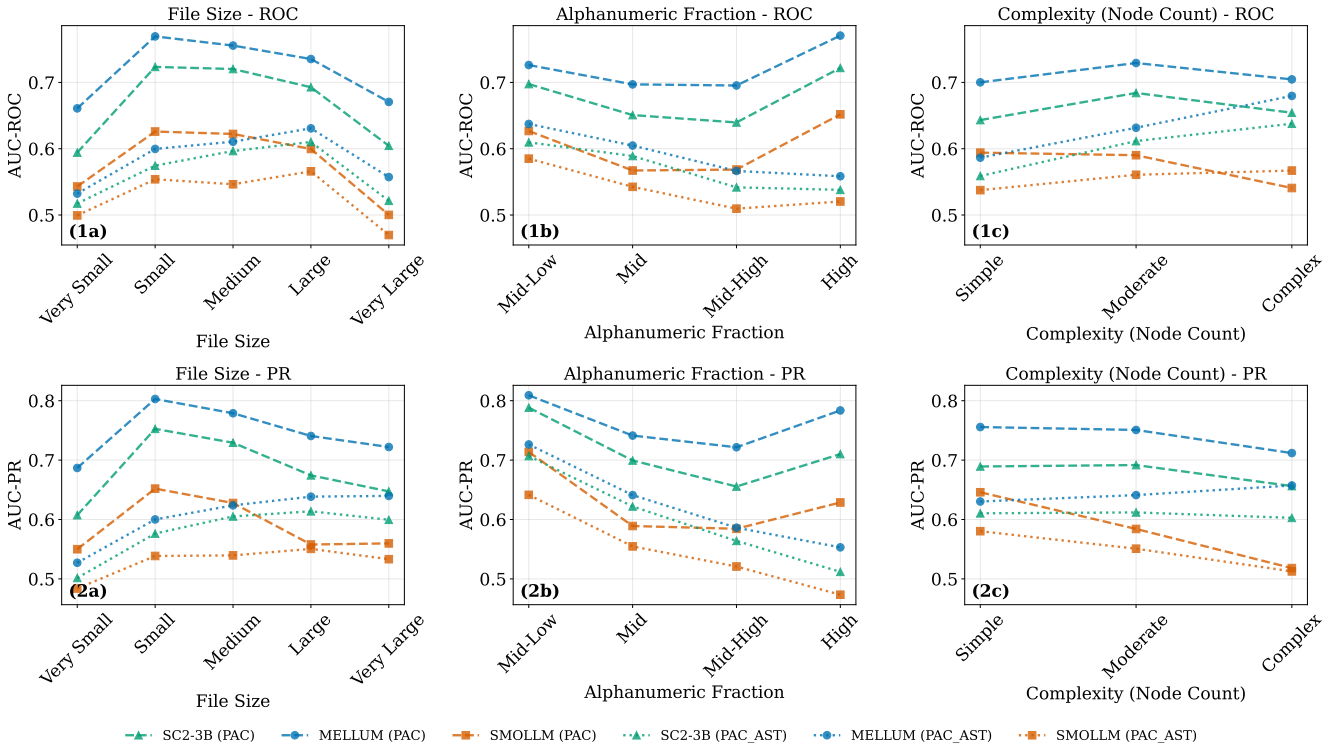

Figure 3: PAC vs. PAC-AST stratified results for ME vs. NM.

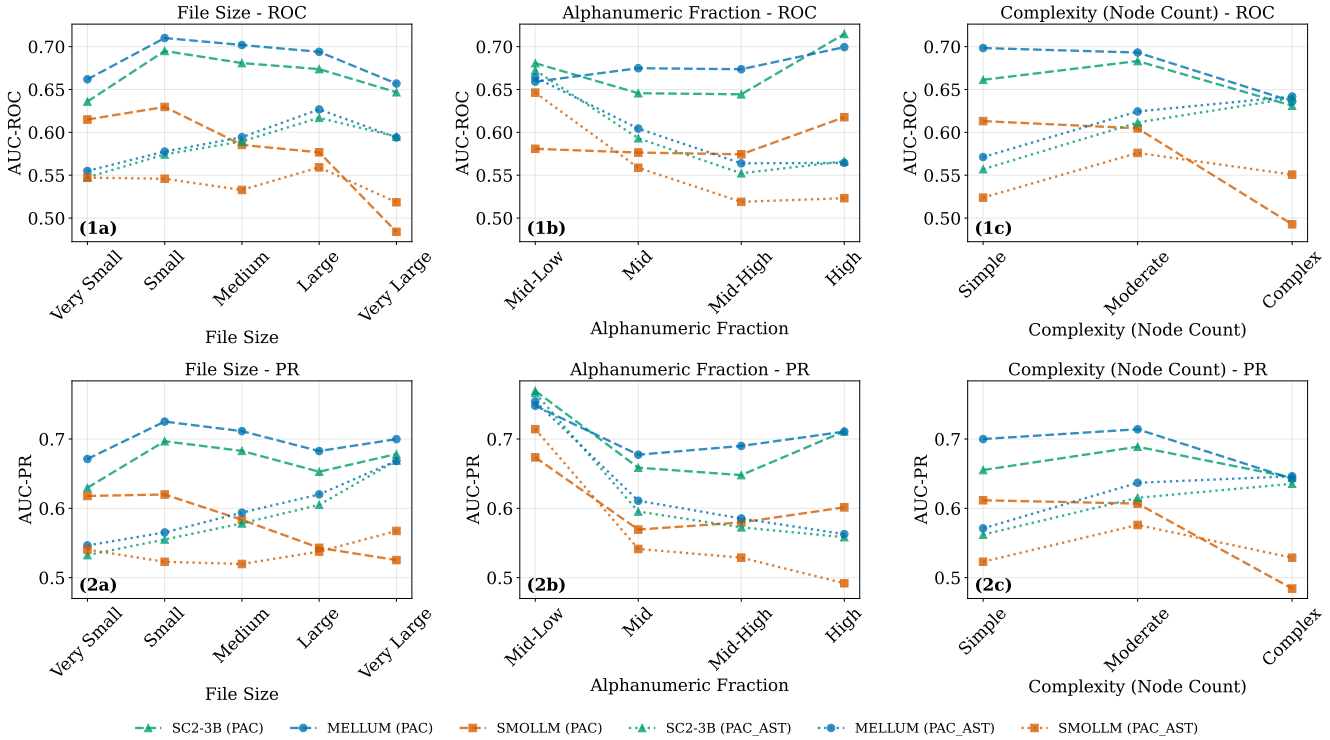

Figure 4: PAC vs. PAC-AST stratified results for MN vs. NM.

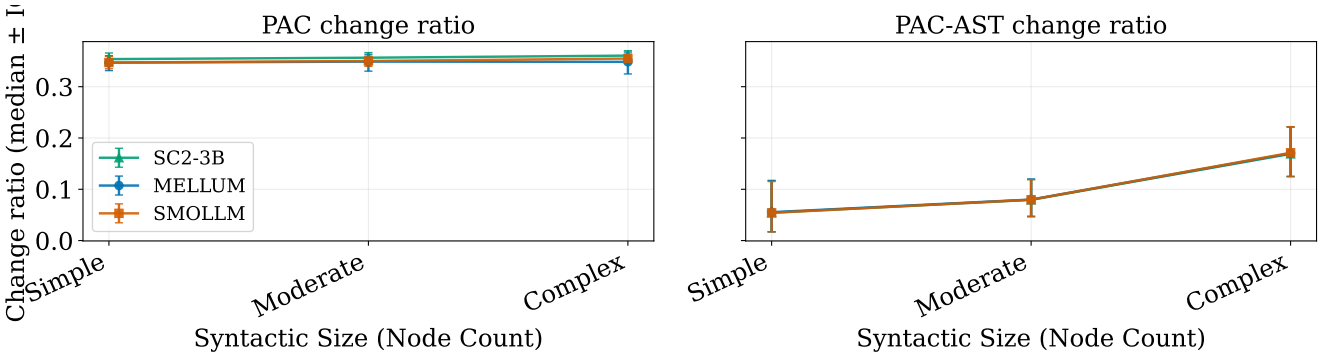

Figure 5: Effective change ratio (median ± IQR) across syntactic size buckets for token-swap PAC and PAC-AST.

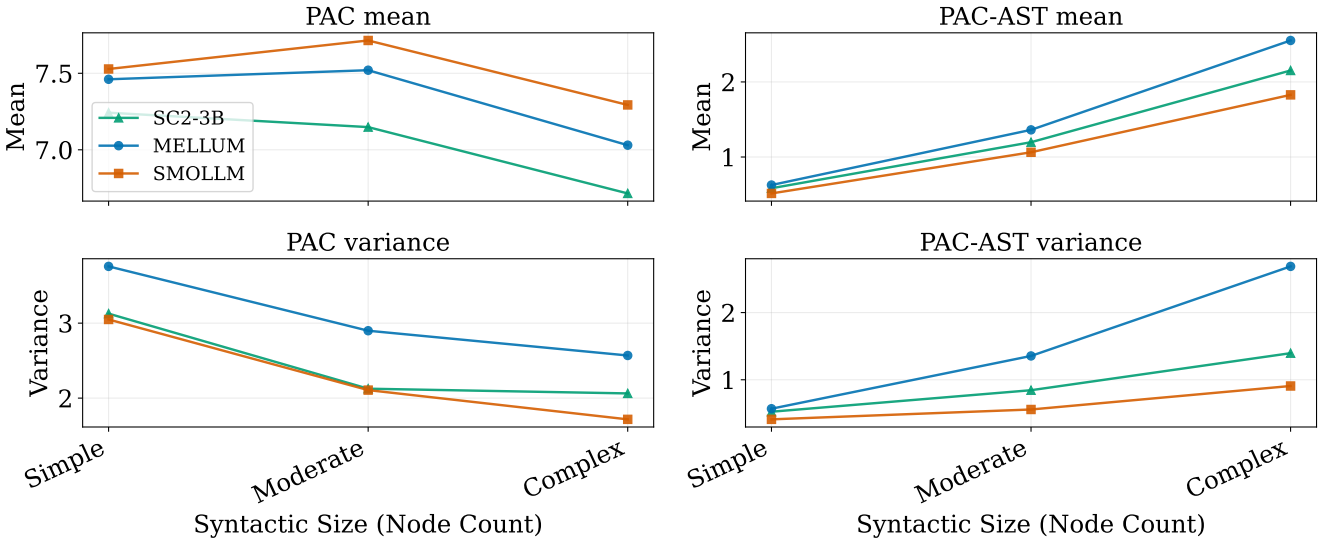

Figure 6: Mean and variance of the calibrated PAC score by syntactic size bucket for PAC and PAC-AST.