Code Membership Inference for Detecting Unauthorized Data Use in Code Pre-trained Language Models

Sheng Zhang, Hui Li*, Rongrong Ji
Key Laboratory of Multimedia Trusted Perception and Efficient Computing
Ministry of Education of China, Xiamen University
sheng@stu.xmu.edu.com, {hui,rrji}@xmu.edu.com

Abstract

Code pre-trained language models (CPLMs) have received great attention since they can benefit various tasks that facilitate software development and maintenance. However, CPLMs are trained on massive open-source code, raising concerns about potential data infringement. This paper launches the study of detecting unauthorized code use in CPLMs, i.e., Code Membership Inference (CMI) task. We design a framework BUZZER for different settings of CMI. BUZZER deploys several inference techniques, including signal extraction from pre-training tasks, hard-to-learn sample calibration and weighted inference, to identify code membership status accurately. Extensive experiments show that CMI can be achieved with high accuracy using BUZZER. Hence, BUZZER can serve as a CMI tool and help protect intellectual property rights. The implementation of BUZZER is available at: https://github.com/KDEGroup/Buzzer.

1 Introduction

Recently, various code pre-trained language models (CPLMs) like CodeBERT (Feng et al., 2020) and Code Llama (Rozière et al., 2023) have sprung up and shown strong capabilities. CPLMs are pre-trained over massive code data that is publicly available in platforms like GitHub and StackOverflow. Then, CPLMs can be fine-tuned or directly used for code-related tasks like code refactoring (Liu et al., 2023a) and code search (Wang et al., 2022a) even when the downstream tasks do not have much data, reducing the intellectual burden of developers and facilitating software development and maintenance

However, using code data to train CPLMs may cause patent infringement and legal violations. GitHub recently introduced a programming tool Copilot[1]. Copilot is powered by OpenAI Codex, a GPT based CPLM. However, Copilot has faced allegations of violating open-source licenses (InfoQ, 2022) since it is trained on code that may be collected from open-source projects. Although it is still under debate whether using open-source code to train CPLMs causes intellectual property infringement, the lawsuit has alerted researchers and companies who work on CPLMs: code data from open-source projects is *not free* training data.

To help protect the intellectual property rights on code data, this paper studies a new task named Code Membership Inference (CMI) for CPLMs which identifies whether a well-trained CPLM used a certain code snippet as its training data. A CMI method for CPLMs can serve as a tool to detect unauthorized data use and provide potential evidence when a lawsuit similar to Copilot's case is filed in the future. We propose a CMI framework BUZZER for detecting unauthorized data use in different settings of CMI. The contributions of this work are summarized as follows:

1. We define two levels of inference for CPLMs: white-box inference and black-box inference. They have different knowledge w.r.t. CPLMs and training data. White-box inference is hard to achieve, but it can help us understand the upper bound of the accuracy of CMI, while black-box inference is more likely to succeed in practice.

2. For the two settings, our proposed BUZZER applies various inference techniques, including signal extraction from pre-training tasks, hard-to-learn sample calibration and weighted inference, to identify code membership status accurately.

3. We have conducted CMI on representative CPLMs. Experimental results show that BUZZER can achieve promising accuracy. Additionally, we find that the accuracy of

[*] Corresponding Author.
[1] https://github.com/features/copilot

BUZZER in black-box inference is not much worse than that in white-box inference, showing that CMI can be achieved with high accuracy in practice.

2 Related Work

2.1 Code Pre-trained Language Model

Prevalent CPLMs (Feng et al., 2020; Guo et al., 2021, 2022) typically adopt a multi-layer Transformer architecture (Vaswani et al., 2017) with N Transformer blocks (we call them hidden layers in this paper). Given a code snippet c, CPLM encodes it into high-dimension representation vectors. Before feeding c into the CPLM, it is natural to tokenize c into a series of tokens $\{c_1 \cdots c_n\}$. Then, tokens will be encoded by the CPLM into representation vectors $\{\mathbf{h}_1, \cdots, \mathbf{h}_n\}$ that can be further used in downstream code-related tasks. Note that, CPLMs can also encode the corresponding descriptions $\{o_1 \cdots o_m\}$ of c (e.g., method comment) written in natural language into token representations $\{\mathbf{r}_1, \cdots, \mathbf{r}_m\}$ (Feng et al., 2020).

According to their model architectures, recent works can be categorized into three types:

- **Encoder Based Models.** Encoder-only CPLMs typically follow the design of BERT (Liu et al., 2019). CodeBERT (Feng et al., 2020) uses masked language modeling and replaced token detection tasks for pre-training. GraphCodeBERT (Guo et al., 2021) models code data from a structural perspective and it uses edge prediction and node alignment as the pre-training tasks.

- **Decoder Based Models.** Decoder based CPLMs only utilize multi-layer transformer decoders, and they are known for their enhanced generalization capabilities in generative tasks (Liu et al., 2023b). IntelliCode (Svyatkovskiy et al., 2020) and CodeGPT (Lu et al., 2021) follow the objective of GPT-2, employing the next token prediction task for pre-training. Based on Llama2 (Touvron et al., 2023), CodeLlama (Rozière et al., 2023) expands the model input length to 16k tokens and performs the pre-training task of fill-in-the-middle (Bavarian et al., 2022). DeepSeek-Coder (Guo et al., 2023) is pre-trained on a vast dataset containing 87 programming languages with dependency parsing and repo-level deduplication. It undergoes training for both the next token prediction and fill-in-the-middle tasks.

- **Encoder-Decoder Based Models.** Encoder-Decoder based CPLMs contain both encoder and decoder in transformer, and they perform well for both understanding and generation tasks. Jiang et al. propose TreeBERT (Jiang et al., 2021), which utilizes tree structure of abstract syntax trees (ASTs) and models them as a set of composition paths to enhance the understanding of code data. SPT-Code (Niu et al., 2022) leverages ASTs to enhance semantic representation. It improves the generation ability of CPLMs by setting up special pre-training tasks, including Code-AST prediction, Masked Sequence to Sequence (MASS) (Song et al., 2019), and method name generation. CodeT5 (Wang et al., 2021) employs a unified framework to seamlessly support both code understanding and generation tasks, and it allows multi-task learning. UniXcoder (Guo et al., 2022) utilizes prefix adapters to control the model behaviors and leverages multimodal data for enhancing code comprehension and code generation tasks.

2.2 Membership Inference

Membership Inference (MI) (Hu et al., 2022) aims to ascertain whether a given data record is part of a particular dataset used to train a specific model.

Shokri et al. (Shokri et al., 2017) study membership inference by utilizing multiple shadow models to mimic the target model. Following Shokri, Salem et al. (Salem et al., 2019) relax the restrictions by reduce the number of shadow models and perform membership inference with less knowledge of member data Yeom et al. (Yeom et al., 2018) investigate the role of overfitting in membership inference for popular machine learning algorithms. Li et al. (Li and Zhang, 2021) perform membership inference by only accessing the final predicted label, instead of acquiring the logits or probabilities.

For language models, Song et al. (Song and Raghunathan, 2020) study membership inference for word embedding models by calculating the average similarity in a sliding window. Mahloujifar et al. (Mahloujifar et al., 2021) leverage the semantic relationships preserved by word embeddings to identify special word pairs. Jagannatha et al. (Jagannatha et al., 2021) investigate the risk of training

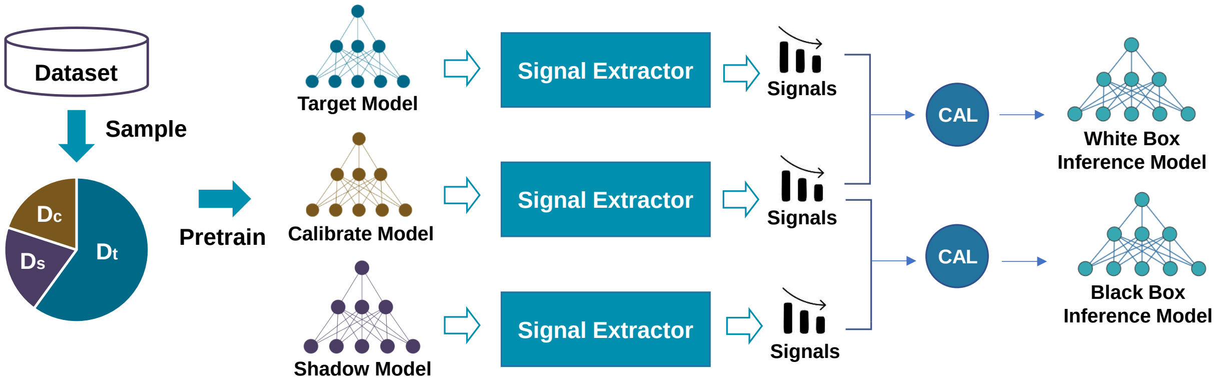

Figure 1: Overview of our BUZZER framework. Firstly, it samples three disjoint datasets, D_t, D_s and D_c, to construct target, shadow and calibrated models, respectively. After that, it extracts model signals with calibration and trains white-box and black-box classifiers for CMI.

data leakage in clinical models. Mireshghallah et al. (Mireshghallah et al., 2022) introduce a reference model and give the determination based on the likelihood ratio threshold. CMI is a specific type of membership inference (MI) (Hu et al., 2022). Yang et al. (Yang et al., 2023) studies the code membership inference task for auto-regressive models. Their work is closely related to ours. Differently, our method can be applied to CPLMs with other architectures in addition to auto-regressive CPLMs.

3 Code Membership Inference in CPLMs

3.1 Task Definition

We first give the definition of the CMI task:

Definition 1 (Code Membership Inference) *Given a tokenized code snippet* $\{c_1, c_2, \cdots, c_n\}$, *a tokenized corresponding natural language descriptions* $\{r_1, r_2, \cdots, r_m\}$ *and the target CPLM* $\mathcal{M}$, *the adversary adopts an inference model to determine whether* c *is in the training data of* $\mathcal{M}$.

Instead of giving "hard prediction", the inference model can output a continuous *confidence score* indicating the probability of c being the code member data. Then, the adversary uses a threshold θ to yield the prediction:

$$\mathcal{A}(c) = \mathbb{1}\left[\mathcal{I}(c) > \theta\right], \tag{1}$$

where $\mathbb{1}$ is the indicator function, θ is a chosen threshold, $\mathcal{I}(\cdot)$ is the inference model that produces the confidence score, and $\mathcal{A}(\cdot)$ is the membership indicator.

3.2 Knowledge Level

The knowledge of the adversary on $\mathcal{M}$ is critical to the success of CMI. We define two inference settings with different knowledge levels:

1. **White-Box Inference:** The adversary has complete knowledge of $\mathcal{M}$ (e.g., model architecture, training objectives, and the trained model parameters). Moreover, the adversary can access a considerable amount of the training data, converting the problem into a supervised classification problem. In practice, this is hard to achieve from the outside model provider.

2. **Black-Box Inference:** The adversary knows the core architecture (e.g., Transformer) and pre-training objectives of $\mathcal{M}$. Such information is typically available via public technical reports (e.g., technical reports of CodeLlama and CodeT5 are publicly available). Hence, compared to white-box inference, black-box inference is a more practical setting of CMI.

3.3 Our Proposed BUZZER

This section illustrates the details of BUZZER. As depicted in Fig. 1, BUZZER is designed for handling both white-box and black-box settings of CMI. First, it samples disjoint datasets to construct target, shadow and calibration models. Then, it extracts model signals with calibration and trains white-box and black-box classifiers.

3.3.1 Overview of Two Types of CMI

White-Box Inference. Taking advantage of the prior knowledge on the considerable amount of training data, the adversary can train an inference model to infer membership status. The adversary can mix known code member data and other code data (code non-member data) that is very unlikely to be the code member data to construct the inference model's training data. Code non-member data can be sampled from the population of the dataset $\mathcal{D}$ that are not included in the description of the training data sources of $\mathcal{M}$. This way, white-box inference becomes a binary classification problem.

The next question is how to define the behavior of $\mathcal{M}$ w.r.t. a certain code snippet c. Recall that the input tokens of c are first encoded by the embedding layer and the embeddings are passed to the first hidden layer. There are multiple hidden layers in a CPLM, and each of them applies a non-linear function on the inputs from preceding layer. The overall forward propagation process can be described as follows:

$$\mathbf{H}_{i+1} = \text{Layer}_i\left(\mathbf{H}_i\right), \tag{2}$$

where $\mathbf{H}_i$ refers to the output of the i-th hidden layer and $\text{Layer}_i(\cdot)$ indicates the i-th hidden layer. Hidden layers are key components that enable the CPLM to understand code data. Therefore, after feeding c to $\mathcal{M}$, we can regards the outputs of hidden layers as the behavior of $\mathcal{M}$.

In white-box inference, BUZZER first sample a target dataset D_t and a calibration dataset D_c to construct a target model and a calibration model (see Sec. 3.3.3), respectively. For an interested code snippet c, a signal extractor (see Sec. 3.3.2) undertakes several pre-training tasks of the target CPLM's on the target model and the calibration model to extract signals. These signals will then be fed into a inference model to derive the CMI outcome.

Black-Box Inference. In black-box inference, the adversary lacks access to the $\mathcal{M}$'s member data, raising a challenge for CMI: the adversary lacks labeled member and non-member data for supervised binary classification as in white-box inference.

To overcome this problem, BUZZER samples a shadow dataset D_s to train a shadow model. The shadow model is designed to imitate $\mathcal{M}$ with similar structure and training algorithms. The adversary knows the member data (i.e., D_s) and non-member data of the shadow model. Therefore, the adversary can use the shadow model to replace the target model in the black-box setting and infer code member status. For an interested code snippet c, the signal extractor undertakes several pre-training tasks of the target CPLM's on the shadow model and the calibration model to extract signals. These signals will then be fed into the inference model for the CMI task.

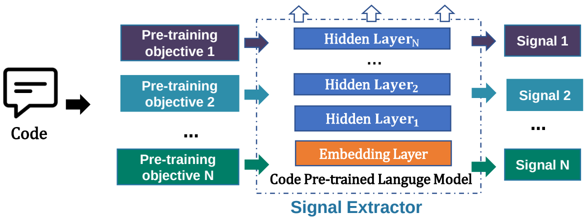

Figure 2: Overview of the signal extractor. It returns signals w.r.t. the pre-training tasks.

Difference between White-box Inference and Black-box Inferencec: The primary distinction lies in the white-box's ability to access the training data of $\mathcal{M}$, while the black-box setting lacks such access. During the training phase, the white-box inference employs both member and non-member data of $\mathcal{M}$ to train its inference model, whereas the black-box inference utilizes member and non-member data of the shadow model. During the testing phase, both white-box and black-box settings conduct tests on member and non-member data of $\mathcal{M}$. More details can be found in 4.1.4.

3.3.2 Signal Extractor

The signal extractor captures CPLM's behavior when encountering member or non-member data. It extracts signals, which serves as input features to subsequent inference models, from a code snippet.

One question is how to define the signal of CPLMs, which should be highly correlated with both member samples and non-member samples. An intuitive approach is using task-specific loss values as signals. As the CPLM has captured member data well, encountering member data could results in lower loss of CPLM. Likewise, the loss values for non-member data generally turn to be lower.

Fig. 2 depicts the design of the signal extractor. For a code snippet, the signal extractor undertakes several pre-training tasks, which are consistent with the target CPLM's original pre-training tasks, on the target/shadow/calibration model. Through these tasks, we acquire the loss values w.r.t. input code snippet which serves as the signals. These signals will be used as input features for the subsequent inference model.

3.3.3 Calibration Model

Non-member samples can still produce strong signals indicating a high probability of being member data, whereas a member sample may yield the opposite result, since they are over-represented or under-represented in the data distribution (Watson et al., 2022). Hence, we design a calibration model to tackle such hard-to-learn samples. The calibration model adopts the same model architecture as the target CPLM. The calibration model is utilized to determine sample difficulty and it is pre-trained using a dataset, which is disjoint from both the shadow model dataset and the target model dataset, and the same pre-training tasks as the target CPLM.

The hard-to-learn calibration process ("CAL" in Fig. 1) is illustrated in Eq. 3, where $signal_i^{cal}$ represents the final calibration value of i_{th} sample and $signal_i^{t}$ denotes the signal of the target model (white-box inference) or the shadow model (black-box inference), $signal_i^{c}$ indicates the signal of the calibration model, and ϵ prevents division by zero.

$$signal_i^{cal} = \frac{signal_i^{t}}{signal_i^{c} + \epsilon}. \tag{3}$$

3.3.4 Weighted Inference Model

The inference model incorporates multiple signals from the signal extractor to generate the final prediction score. A higher score assigned by the inference model to a code snippet indicates higher likelihood of belonging to the target model's member data. The inference model is designed to take signals from two types of pre-training tasks:

- **Signals from generative pre-training tasks**: The signals generated from generative pre-training tasks (e.g., Bimodal Dual Generation, BDG) (Wang et al., 2021). For these signals, we utilize a one-layer self-attention based network with 12 heads to learn the feature.

- **Signals from non-generative pre-training tasks**: The signals generated from non-generative pre-training tasks (e.g., Masked Language Modeling, MLM) (Feng et al., 2020). For these signals, we utilize a three-layer multi-layer perceptron based network to learn the feature.

Consider a target CPLM with pre-training tasks such as MLM and BDG. Initially, signals are extracted from the loss values of the pre-training tasks. Subsequently, the inference model adopts two different sub-networks, which take different types of signals as input, to predict the confidence scores. For signals generated from generative pre-training tasks (e.g., BDG), sub-networks with the self-attention mechanism are employed. For signals from non-generative pre-training tasks (e.g., MLM), sub-networks with three-layer MLP with ReLU are utilized. Finally, BUZZER weights the confidence scores of different sub-networks as the final confidence score in Eq. 1.

The loss function of the inference model is illustrated in Eq. 4, which is designed to maximize the model output of a member sample and minimize that of a non-member sample.

$$\mathcal{L}(\Theta, m, n) = \alpha - C(m) + C(n), \tag{4}$$

where m and n represent the member and non-member sample, respectively. $C(m)$ is the output of inference model and α is a hyper-parameter.

4 Experiments

4.1 Settings

4.1.1 Evaluation Metrics

We adopt Area Under the Curve (AUC) as the main evaluation metrics, which assesses a model's capability to differentiate between positive and negative samples. AUC is widely used in evaluating MI (Li and Zhang, 2021; Mireshghallah et al., 2022; Zhang et al., 2021; Wang et al., 2022b). We also consider True Positive Rates (TPR) at low False Positive Rates (FPR) as the evaluate metric (Carlini et al., 2022). Specifically, we compare TPR values of different methods when the target FPR values are low (1%, 0.1% and 0.01%).

4.1.2 CPLMs

We choose four representative CPLMs as target models, including CodeBERT[2] (Feng et al., 2020), CodeT5[3] (Wang et al., 2021), DeepseekCoder[4] (Guo et al., 2023) and CodeLlama[5] (Rozière et al., 2023). CodeBERT is a BERT based bimodal CPLM. CodeT5 is an encoder-decoder based CPLM. DeepseekCoder is a decoder-only model and we adopt *deepseek-coder-1.3b-base* with 1.3B parameters. CodeLlama is a decoder-only model based on Llama 2 and we adopt *codellama-7b-base* with 7B parameters. For CodeBERT and CodeT5, we pre-train them from scratch to generate target models, resulting CodeBERT with 125M parameters and CodeT5 with 220M parameters. For larger

[2] https://github.com/microsoft/CodeBERT

[3] https://github.com/salesforce/CodeT5

[4] https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base

[5] https://huggingface.co/codellama/CodeLlama-7b-hf

CPLMs DeepseekCoder and CodeLlama, we continue training based on their released models to generate target models.

4.1.3 CMI Baselines

We compare Buzzer with three CMI baselines:

- **FastText** (Joulin et al., 2017): It is a text classification library, which can be used to intuitively demonstrate whether there are distributional differences between member data and non-member data. We train FastText with the member data and the non-member data of $\mathcal{M}$, and use it to directly assess whether a code snippet is member or non-member data.

- **Perturbation** (Carlini et al., 2021): It perturbs a code snippet by converting case and calculates the L2 distance before and after the transformation to determine whether it is member data.

- **Perplexity** (Carlini et al., 2021; Inan et al., 2021; Oh et al., 2023): It calculates the perplexity of an interested data record to determine whether it is member data. The intuition behind it is that the member data may have lower perplexity. Perplexity is commonly used for the CMI task.

4.1.4 Data

For CodeBERT and CodeT5, we choose CSN[6] (Husain et al., 2019) dataset since their authors pretrain CodeBERT and CodeT5 over CSN. CSN dataset contains over 6 million code snippets from open-source projects on GitHub, spanning six programming languages (Python, Java, JavaScript, Go, Ruby, and PHP). Due to limited computational resource, we only use python code snippets of CSN. Code snippets are associated with metadata such code description written in natural language. We sample disjoint segments of CSN to pre-train target, shadow and calibration models. Specifically, we sample 100,000 data records for pre-training the target model, 50,000 for pre-training the shadow model and the calibration model, respectively. For testing, we sample 10,000 member data records and 10,000 non-member data records.

For DeepseekCoder and CodeLlama, we adopt Magicoder-Evol-Instruct-110k (MEI)[7] (Wei et al., 2023). Noted that MEI is generated by GPT-4 and the data leakage issue can be avoided. Specifically, we sample 30,000 data records of MEI for training the target model, 20,000 for training the shadow model and the calibration model, respectively. For testing, we sample 5,000 member data records and 5,000 non-member data records.

Table 1: Performance of white-box and black-box inference. “-” indicates the values are almost zero.

Method	AUC	TPR		
		0.01%FPR	0.10%FPR	1.00%FPR
FastText	0.500	-	-	-
CodeBERT$_{perb}$	0.505	0.00%	0.12%	0.90%
CodeT5$_{perb}$	0.499	0.03%	0.15%	1.01%
DeepseekCoder$_{ppl}$	0.501	0.30%	0.30%	1.21%
CodeLlama$_{ppl}$	0.670	2.23%	2.23%	8.50%
CodeBERT$_{wb}$	0.603	0.06%	0.28%	2.36%
CodeT5$_{wb}$	0.869	0.04%	1.42%	12.16%
DeepseekCoder$_{wb}$	0.722	1.16%	5.05%	16.33%
CodeLlama$_{wb}$	0.980	22.57%	43.01%	83.80%
CodeBERT$_{bb}$	0.602	0.03%	0.26%	2.42%
CodeT5$_{bb}$	0.859	0.14%	1.65%	12.06%
DeepseekCoder$_{bb}$	0.721	1.08%	5.01%	16.33%
CodeLlama$_{bb}$	0.979	21.90%	41.07%	83.45%

4.1.5 Environment and Hyper-Parameters

We run the experiments on a machine with two Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz, 256 GB main memory and eight NVIDIA GeForce RTX 3090. We implement CodeBERT and CodeT5 following their original papers since only their pretraining implementations are not public available. For training CodeLlama, we utilize deepspeed[8] and ZERO 1 optimization with cpu offload (Rajbhandari et al., 2020). We set the batch size to 64 and learning rate to 5e-5. Other hyper-parameters are set according to original papers.

4.2 Experimental Results

4.2.1 Overall Performance

Tab. 1 provides the overall results. The abbreviations bb and wb stand for black-box Buzzer and white-box Buzzer, respectively. $perb$ and ppl indicate Perturbation and Perplexity, respectively. From Tab. 1, we have the following findings:

- Member data and non-member data can not be easily separated according to code features (i.e., data distribution), as evidenced by the results of FastText: it achieves an AUC score close to 0.5.

[6] https://github.com/github/CodeSearchNet

[7] https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K

[8] https://github.com/microsoft/DeepSpeed

- BUZZER consistently shows superior performance than baselines FastText, Perturbation and Perplexity, showing the effectiveness of BUZZER.

- The AUC of white-box inference is not much higher than that of black-box inference. Hence, we can conclude that knowing the distribution of the training dataset of the target model has a relatively minor impact on the inference accuracy. *In other words, CMI in the black-box setting can be achieved with high accuracy.*

- The AUC scores for CodeT5 (~0.8), DeepseekCoder (~0.7) and CodeLlama (~0.9) are much higher than that of CodeBERT (~0.6). A possible reason is that they have different model structures and parameter sizes. Recent works have found larger language models tend to over-memorized training data (member data) than smaller language models (Tirumala et al., 2022; Carlini et al., 2023), which may demonstrate why the AUC for CodeBERT, the smallest CPLM, is lowest.

- To investigate whether BUZZER suffers from high false positive rate, we show TPR under different FPR (0.01%, 0.1%, 1%) in Tab. 1. We can see that BUZZER overcomes the high FPR issue, a common problem in existing MI works (Watson et al., 2022) since the TPR of BUZZER is much higher than TPR of the baselines under a low FPR. BUZZER shows much higher TPR on larger CPLMs DeepseekCoder and CodeLlama. The over-memorizing characteristic of larger language models (Tirumala et al., 2022; Carlini et al., 2023) may be the reason.

4.2.2 Impact of Data Characteristics

Next, we study the impacts of different code characteristics on the inference results. In other words, we are interested in the factors that affect how BUZZER makes membership status predictions. We investigate three common code features:

- **Code Length:** Code length refers to the length of a code snippet. Longer code snippets can provide more information.

- **Depth of AST:** The abstract syntax tree is an important feature that distinguishes code from natural language. The depth of the code abstract syntax tree may affect the inference results.

- **Node Number of AST:** An AST node represents a fundamental component of the structure of a code snippet. Thus the number of AST node may affect the inference results.

Table 2: Impact of the calibration model (AUC).

Method	White Box		Black Box	
	w/ cal	w/o cal	w/ cal	w/o cal
CodeBERT	0.603	0.523	0.602	0.524
CodeT5	0.869	0.732	0.859	0.719
DeepseekCoder	0.722	0.514	0.721	0.514
CodeLlama	0.980	0.817	0.979	0.809

Fig. 3 reports the distributions of the confidence scores of code snippets predicted by CodeBERT$_{wb}$ and CodeT5$_{wb}$. Due to page limit, we only show the results of CodeBERT and CodeT5. First, we obtain the values for the three features of each code snippet. Next, we group the code data into intervals with equal length, and arrange intervals in ascending order based on the corresponding feature. The x axis of Fig. 3 represents the intervals. The left y axis is the number of code samples in each interval and the right y axis denotes the confidence score. We normalize the scores to a range between 0 and 1 via min-max normalization. The bar charts in Fig. 3 represents the number of samples in each group, while the line graphs show the average confidence scores of each group. Subsequently, we examine the scores assigned by the inference model to different groups and analyze whether any specific patterns emerge across the intervals.

From Fig. 3, we can observe the long-tail distributions of code snippets grouped by the three code features. If we consider both confidence scores and number of code snippets for each interval, then we can find that, for CodeT5, the predicted confidence scores are positively correlated with code length, depth of AST, and number of AST nodes. Differently, for CodeBERT, they are negatively correlated. The differences are possibly caused by their model structures. For CodeT5, it is encoder-decoder structure, which consists of bidirectional attention and unidirectional attention mechanisms. When the sequence is long, tokens closer to the beginning of the sequence in the decoder receive more attention, leading to stronger training signals.

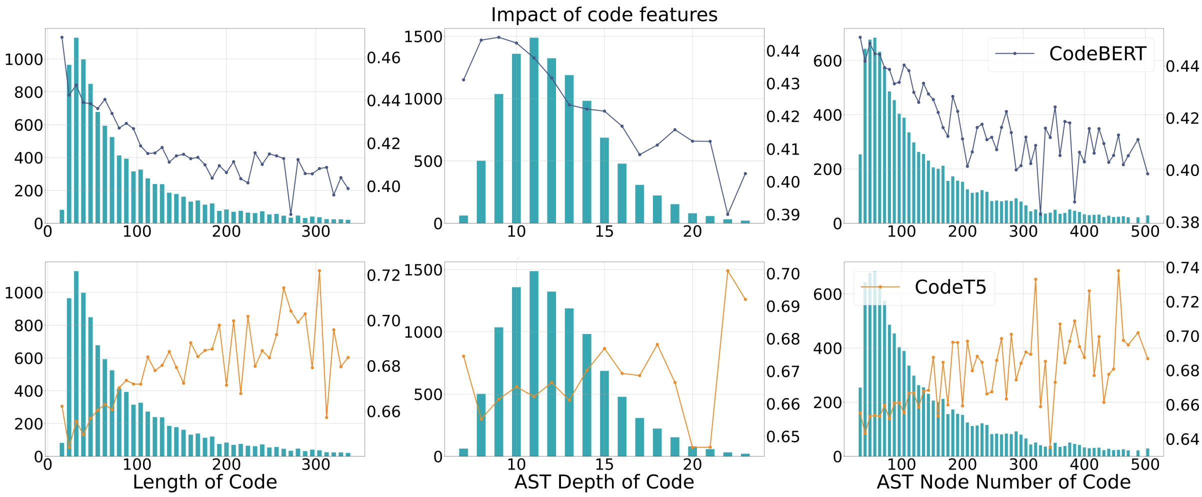

Figure 3: Impact of different code features.

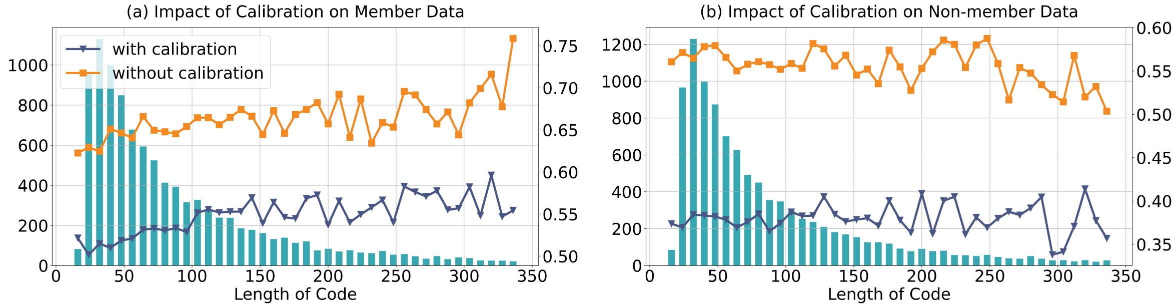

Figure 4: Impact of calibration.

Table 3: Impact of Training Data Size (AUC).

Model	Data Number		
	20K	10K	5K
DeepseekCoder$_{bb}$	0.721	0.696	0.657

Table 4: HumanEval (pass@1) of the target model.

Method	Before	After
DeepseekCoder	0.34	0.36
CodeLlama	0.30	0.42

4.2.3 Analysis of Calibration Model

Tab. 2 displays the inference results with and without the calibration model. It is evident that the calibration model can significantly improve the CMI performance. Specifically, for CodeBERT, it increases the AUC score by 0.08, and for CodeT5, by 0.14. For DeepseekCoder and CodeLlama, the effect of calibration on black-box inference is more significant. The calibration model is effective in both white-box inference and black-box inference.

Fig. 4 further shows the improvements brought by the calibration model. We bucketize the data w.r.t. code length in a similar way as Sec. 4.2.2. Note that a higher AUC score indicates that the model can better separate member and non-member data, i.e., the gap between the confidence scores of member and non-member data is larger. In Fig. 4 (a), the minimum score of member data with calibration is around 0.52 (the blue curve in Fig. 4 (a)), while the maximum score of non-member data with calibration is around 0.41 (the blue curve in Fig. 4 (b)). The gap with calibration is 0.11. The minimum score of member data without calibration is around 0.62 (the orange curve in Fig. 4 (a)), while the maximum score of non-member data without calibration is around 0.59 (the orange curve in Fig. 4 (b)). The gap without calibration is 0.03, which is smaller than the gap with calibration (0.11). Hence, we can conclude that calibration increase the gap between member and non-member data, resulting in higher AUC.

4.2.4 Impact of Training Data Size

To investigate how the size of training data for constructing BUZZER affects the performance, we report the effects of different data size on BUZZER over DeepseekCoder in black-box setting in Tab. 3. Note that the results of DeepseekCoder$_{bb}$ in Tab. 1 is reported using default training data size 20K. From Tab. 3, we can see that, as the size of training data decreases, AUC of BUZZER declines. However, when the training data size is small (5K), BUZZER can still achieve an AUC score of 0.657, showing that it can be effective even when not much training data is available.

4.2.5 Effectiveness of Training Larger CPLMs

As we continue training with additional data over the pre-trained DeepseekCoder and CodeLlama to generate the target models, it is essential to investigate how the training affects the performance of the two large CPLMs. Tab. 4 shows the performance of DeepseekCoder and CodeLlama on the HumanEval benchmark (Chen et al., 2021), a popular benchmark to evaluate code generation. We can that see that the pass@1 rate has increased after our training, indicating that the process of constructing target models positively affects DeepseekCoder and CodeLlama.

5 Conclusion

In this paper, we study CMI for authenticating data compliance in CPLMs and propose a framework BUZZER for inferring code membership. BUZZER achieves promising results on various CPLMs as shown in the experiments. BUZZER can serve as a CMI tool and help protect the intellectual property rights. In the future, we plan to further improve the generalization ability of BUZZER to make it more practical. We will also explore the idea of this work on other multimodal pre-trained language models beyond CPLMs.

6 Limitations

We study CMI using public code data that is not originally designed for this task. In practice, the code data that their owners care about may not be publicly available, making it difficult to collect them for the study of CMI. For such cases, it is difficult to assess the performance of BUZZER based on the results reported in this paper.

Acknowledgments

This work was partially supported by National Science and Technology Major Project (No. 2022ZD0118201), Natural Science Foundation of Xiamen, China (No. 3502Z202471028) and National Natural Science Foundation of China (No. 62002303, 42171456).

References

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv Preprint*. https://arxiv.org/abs/2207.14255.

Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramèr. 2022. Membership inference attacks from first principles. In *IEEE Symposium on Security and Privacy*, pages 1897–1914.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2023. Quantifying memorization across neural language models. In *ICLR*. https://openreview.net/pdf?id=TatRHT_1cK.

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 *USENIX Security Symposium*, pages 2633–2650.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv Preprint*. https://arxiv.org/abs/2107.03374.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *EMNLP (Findings)*, pages 1536–1547.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *ACL*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *ICLR*. https://openreview.net/pdf?id=jLoC4ez43PZ.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2023. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv Preprint*. https://arxiv.org/abs/2401.14196.

Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S. Yu, and Xuyun Zhang. 2022. Membership inference attacks on machine learning: A survey. *ACM Comput. Surv.*, 54(11s):235:1–235:37.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv Preprint*. https://arxiv.org/abs/1909.09436.

Huseyin A. Inan, Osman Ramadan, Lukas Wutschitz, Daniel Jones, Victor Rühle, James Withers, and Robert Sim. 2021. Training data leakage analysis in language models. *arXiv Preprint*. https://arxiv.org/abs/2101.05405.

InfoQ. 2022. First open source copyright lawsuit challenges github copilot. https://www.infoq.com/news/2022/11/lawsuit-github-copilot/.

Abhyuday Jagannatha, Bhanu Pratap Singh Rawat, and Hong Yu. 2021. Membership inference attack susceptibility of clinical language models. *arXiv Preprint*. https://arxiv.org/abs/2104.08305.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *UAI*, volume 161, pages 54–63.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomás Mikolov. 2017. Bag of tricks for efficient text classification. In *EACL*, pages 427–431.

Zheng Li and Yang Zhang. 2021. Membership leakage in label-only exposures. In *CCS*, pages 880–895.

Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023a. Refbert: A two-stage pre-trained framework for automatic rename refactoring. In *ISSTA*, pages 740–752.

Xin Liu, Daniel McDuff, Geza Kovacs, Isaac R. Galatzer-Levy, Jacob E. Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak N. Patel. 2023b. Large language models are few-shot health learners. *arXiv Preprint*. https://arxiv.org/abs/2305.15525.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *arXiv Preprint*. https://arxiv.org/abs/1907.11692.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*.

Saeed Mahloujifar, Huseyin A Inan, Melissa Chase, Esha Ghosh, and Marcello Hasegawa. 2021. Membership inference on word embedding and beyond. *arXiv Preprint*. https://arxiv.org/abs/2106.11384.

Fatemehsadat Mireshghallah, Kartik Goyal, Archit Uniyal, Taylor Berg-Kirkpatrick, and Reza Shokri. 2022. Quantifying privacy risks of masked language models using membership inference attacks. In *EMNLP*, pages 8332–8347.

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Sptcode: Sequence-to-sequence pre-training for learning source code representations. In *ICSE*, pages 1–13.

Myung Gyo Oh, Leo Hyun Park, Jaeuk Kim, Jaewoo Park, and Taekyoung Kwon. 2023. Membership inference attacks with token-level deduplication on korean language models. *IEEE Access*, 11:10207–10217.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: memory optimizations toward training trillion parameter models. In *SC*, page 20.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *arXiv Preprint*. https://arxiv.org/abs/2308.12950.

Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. 2019. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *NDSS*.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, pages 3–18.

Congzheng Song and Ananth Raghunathan. 2020. Information leakage in embedding models. In *CCS*, pages 377–390.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: masked sequence to sequence pre-training for language generation. In *ICML*, volume 97, pages 5926–5936.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *ESEC/SIGSOFT FSE*, pages 1433–1443.

Kushal Tirumala, Aram H. Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. 2022. Memorization without overfitting: Analyzing the training dynamics of large language models. In *NeurIPS*, pages 38274–38290.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv Preprint*. https://arxiv.org/abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*, pages 5998–6008.

Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022a. Bridging pre-trained models and downstream tasks for source code understanding. In *ICSE*, pages 287–298.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708.

Zihan Wang, Na Huang, Fei Sun, Pengjie Ren, Zhumin Chen, Hengliang Luo, Maarten de Rijke, and Zhaochun Ren. 2022b. Debiasing learning for membership inference attacks against recommender systems. In *KDD*, pages 1959–1968.

Lauren Watson, Chuan Guo, Graham Cormode, and Alexandre Sablayrolles. 2022. On the importance of difficulty calibration in membership inference attacks. In *ICLR*. https://openreview.net/pdf?id=3eIrli0TwQ.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv Preprint*. https://arxiv.org/abs/2312.02120.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. Gotcha! this model uses my code! evaluating membership leakage risks in code models. *arXiv Preprint*. https://arxiv.org/abs/2310.01166.

Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. 2018. Privacy risk in machine learning: Analyzing the connection to overfitting. In *CSF*, pages 268–282.

Minxing Zhang, Zhaochun Ren, Zihan Wang, Pengjie Ren, Zhumin Chen, Pengfei Hu, and Yang Zhang. 2021. Membership inference attacks against recommender systems. In *CCS*, pages 864–879.