The “Code” of Ethics: A Holistic Audit of AI Code Generators

Wanlun Ma, Yiliao Song, *Member, IEEE*, Minhui Xue, *Member, IEEE*, Sheng Wen, and Yang Xiang, *Fellow, IEEE*

***Abstract*—AI-powered programming language generation (PLG) models have gained increasing attention due to their ability to generate source code of programs in a few seconds with a plain program description. Despite their remarkable performance, many concerns are raised over the potential risks of their development and deployment, such as legal issues of copyright infringement induced by training usage of licensed code, and malicious consequences due to the unregulated use of these models. In this article, we present the first-of-its-kind study to systematically investigate the accountability of PLG models from the perspectives of both model development and deployment. In particular, we develop a holistic framework not only to audit the training data usage of PLG models, but also to identify neural code generated by PLG models as well as determine its attribution to a source model. To this end, we propose using membership inference to audit whether a code snippet used is in the PLG model's training data. In addition, we propose a learning-based method to distinguish between human-written code and neural code. In neural code attribution, through both empirical and theoretical analysis, we show that it is impossible to reliably attribute the generation of one code snippet to one model. We then propose two feasible alternative methods: one is to attribute one neural code snippet to one of the candidate PLG models, and the other is to verify whether a set of neural code snippets can be attributed to a given PLG model. The proposed framework thoroughly examines the accountability of PLG models which are verified by extensive experiments. The implementations of our proposed framework are also encapsulated into a new artifact, named CODEFORENSIC, to foster further research.**

***Index Terms*—AI code generator, programming language generation, accountability.**

I. INTRODUCTION

RECENT advancements in deep learning have powered the development of Programming Language Generation (PLG) models, also known as AI code generators like aiXcoder [1], GitHub Copilot [3], and ChatGPT [66], to support code intelligence. Trained on vast quantities of code datasets collected from open-source repositories, PLG models can generate a program code in response to a code prompt, such as the description of a programming problem or a code context immediately. Despite their remarkable performance, many have raised concerns over the potential risk of their development and deployment [39].

From the aspect of model development, unauthorized use of source code for training PLG models can lead to copyright infringement and potentially harm the intellectual property of code creators. PLG models are capable of generating code that is identical to licensed source code but without the corresponding attribution to the original source coder [39]. Moreover, source code used in PLG models is often subject to strict licenses that impose regulations such as commercial usage, modification, and source attribution requirements. Unauthorized usage of source code in PLG models can raise legal and ethical issues, especially if the generated content is not transformative (i.e., the model outputs similar content to copyrighted training data) [39]. For example, GitHub Copilot has been found to generate large segments of copied code without attribution to licensed source code [27], [73], leading to a lawsuit against it for alleged copyright infringement [2]. OpenAI, the owner of ChatGPT, is also facing potential fines and a ban in Europe if it fails to convince the authorities its data use practices are legal [38]. To mitigate this infringement harm, the U.S. Copyright Office recently launched an initiative to examine copyright law and policy issues over the use of copyrighted materials in generative AI training [4]. We notice that copyright protection for code is more complex and challenging than that for creative works such as text or music, which have had extensive legal and technical measures in place to protect them [39], [75]. Although text and code generation have some similarities in how their models are trained, they have each spawned distinctive case law with different assessments when it comes to fair use of copyrighted material [39]. As a result, it is difficult for any arbitration committee or court to apply standard copyright principles, such as idea and/or expression distinctions in text- and music-related cases, into code-related cases due to the technical sophistication and functionality of software program [17]. Therefore, it remains a complex and open challenge for collecting digital forensics to detect and prevent copyright violations in *code domain*.

The potential misuse of PLG models also raises ethical concerns that require attention, particularly for model deployment [19]. One of the major issues is the generation of false information [20], [80] and artificial hallucinations [9], which can

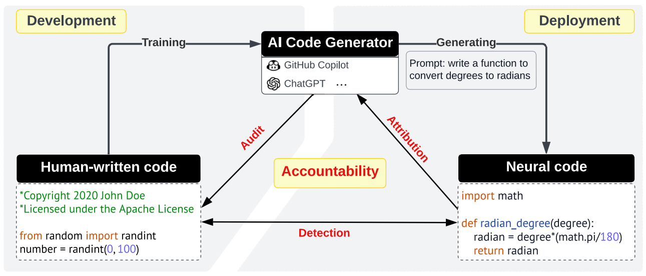
Manuscript received 21 May 2023; revised 2 February 2024; accepted 10 February 2024. Date of publication 20 February 2024; date of current version 4 September 2024. *(Corresponding author: Sheng Wen.)*

Wanlun Ma, Sheng Wen, and Yang Xiang are with the School of Science, Computing and Engineering Technologies, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: wma@swin.edu.au; swen@swin.edu.au; yxiang@swin.edu.au).

Yiliao Song is with the School of Computer and Mathematical Sciences, University of Adelaide, Adelaide SA5005, Australia (e-mail: lia.song@adelaide.edu.au).

Minhui Xue is with the Cybersecurity and Quantum Systems Group, CSIRO's Data61, Eveleigh, NSW 2015, Australia (e-mail: minhuixue@gmail.com).

Digital Object Identifier 10.1109/TDSC.2024.3367737



```
*Copyright 2020 John Doe
*Licensed under the Apache License

from random import randint
number = randint(0, 100)
```

```
import math

def radian_degree(degree):
    radian = degree*(math.pi/180)
    return radian
```

Fig. 1. Overview of our study on accountability of AI code generators.

lead to the spread of misinformation. Not only does this have the potential to harm the public interest, but it can also disrupt the network environment [98]. For instance, to avoid being overwhelmed by the plausible but incorrect AI-generated code, Stack Overflow, a programming-related question-and-answer website recently banned ChatGPT-generated responses [67]. In addition, PLG models can be used for academic plagiarism. Recent research [15] has demonstrated that large PLG models such as GPT-J [89] can accomplish computer science assignments for students without triggering MOSS [76], a widely used academic plagiarism detection tool. To address the potential misuse of PLG models and ensure their responsible use, it is necessary to detect the neural code generated by PLG models and attribute it to the source model.

There is a tangible demand from the regulatory authorities and software industry to develop a holistic framework to examine the accountability of PLG models in terms of auditing training data used by PLG models, detecting neural code generated by PLG models, and attributing it to its source model. Despite much effort made in this area, they solely focus on natural language generation models [34], [41], [47], [62], [82], [87], [97]. Less attention has been paid to emerging PLG models. Additionally, existing studies always investigate model accountability from a single aspect, either the training data [41], [82] or the model output [34], [47], [62], [87], [97]. This prompts the need to examine the accountability of PLG models from a holistic perspective.

Our Work: In this paper, we systematically study the accountability of PLG models from the perspectives of both model development and deployment. In particular, our goal is not only to audit the training data usage of PLG models, but also to detect neural code as well as determine its attribution to a source model. As shown in Fig. 1, we seek to investigate the following research questions (*RQ*s):

- *RQ1: Training Usage Audit.* Given a code snippet, can we determine whether this code snippet is used in the PLG model's training data?
- *RQ2: Neural Code Detection.* Given a code snippet, can we determine whether this code snippet is written by a human or generated by some model?
- *RQ3: Neural Code Attribution.* Given a neural code snippet and a PLG model, can we determine whether the code snippet is generated by the model?

For *training usage audit* (*RQ1*), we perform a membership inference (MI) based on likelihood ratio test (LRT). Experimental results demonstrate the good performance of the LRT-based method. Moreover, considering the limitations of current MI methods, we provide several concrete recommendations from both technical and regulatory perspectives to enhance accountability of PLG models.

For *neural code detection* (*RQ2*), We propose a learning-based classifier to differentiate neural code from human-written code. Our evaluations across five state-of-the-art PLG models and three benchmark datasets reveal the existence of common fingerprints shared across different PLG models. This finding can help software communities detect the neural code generated by PLG models. Moreover, our experiments show that even a well-performing neural text detector cannot effectively distinguish the neural code generated by PLG models, highlighting the critical need to investigate these models and their generated neural code.

We then delve deep to investigate *neural code attribution* (*RQ3*). Here, we aim to determine the attribution of a neural code snippet to a suspected model. We not only empirically demonstrate that it is impossible to reliably solve this one-to-one attribution problem (that is, *it is impossible to reliably attribute ONLY one neural code snippet to a given PLG model*), but provide pioneering theoretical support to this claim.

Hence, we consider two feasible situations where we relax the constraint on either the number of PLG models or the number of neural code snippets. Relaxing the constraint on PLG models allows us to address *attribution classification* using a multi-class classifier to predict the source model for a given neural code snippet. Our experiments reveal distinct fingerprints among different PLG models, enabling us to attribute the neural code snippet to a candidate model. Alternatively, relaxing the constraint on neural code snippets leads to addressing *attribution verification* as a two-sample hypothesis testing problem. Employing the Maximum Mean Discrepancy (MMD) test verifies whether a set of neural code snippets can be attributed to a given PLG model. We theoretically prove that our MMD test can achieve near 100% success with sufficient neural code snippets, approximately 30 as per our empirical studies.

Contributions. Our main contributions are outlined below:

- To the best of our knowledge, we present the first-of-its-kind study to systematically investigate the accountability of PLG models from both model development and deployment perspectives.
- We conduct the first study on the copyright infringement issue of current PLG models, introducing an LRT-based MI method to audit their training usage.
- For neural code detection, we propose a learning-based classifier differentiate neural code from human- written code. Extensive evaluations show the common fingerprints shared among different PLG models, allowing us to effectively detect neural code.
- For neural code attribution, we demonstrate, empirically and theoretically, that it is impossible to reliably attribute the generation of one code snippet to a specific model, shedding light on the challenges of dissecting neural code accountability.
- We propose two feasible alternative methods for neural code attribution: *attribution classification* and *attribution*

TABLE I
SUMMARY OF THE PLG MODELS STUDIED IN OUR WORK

Architecture	Algorithm	ckpt-name	Training dataset	python	java	js	php	ruby	go	c	c#	c++
decoder-only	CodeGen (350M/2B/6B/16B)	CodeGen-NL	ThePile (825G)	◔	◔	◔	◔	◔	◔	◔	◔	◔
		CodeGen-Multi	BigQuery	◑	◑	◑	◔	◔	◑	◑	◔	◑
		CodeGen-Mono	BigPython	●	◑	◑	◔	◔	◑	◑	◔	◑
	InCoder	InCoder-1B/6B	216GB	◑	◑	◑	◑	◔	◑	◑	◔	◑
	PolyCoder	PolyCoder-160M/0.4B/2.7B	249GB	◑	◑	◑	◑	◑	◑	◑	◑	◑
	GPT-Neo	GPT-Neo-125M/1.3B/2.7B	ThePile	◔	◔	◔	◔	◔	◔	◔	◔	◔
encoder-decoder	CodeT5	CodeT5-small/base	CNS+C/C#	◑	◑	◑	◑	◑	◑	◑	◑	○

○: the language is not contained in the training dataset; ◔: the proportion of the language is small (less than 2%);◑: the proportion of the language is about 2% to 50%; ●: the proportion of the language is lager than 50%.

verification. Extensive evaluations demonstrate the effectiveness and robustness of the proposed methods.

- We make our tool and methods publicly available, under the name of CODEFORENSIC,[1] which is a toolkit to characterize neural code accountability.

II. PRELIMINARIES

A. Programming Language Generation Models

In this work, we focus on PLG models that synthesize code from natural language descriptions of programming problems or code context. In this paper, we consider five state-of-the-art code generation models that are publicly available in the open-source community. Details of these models are summarized in Table I.

- *InCoder* [32] Facebook's InCoder is a generative model for program synthesis and editing, using the same architecture as the dense models described in [11]. Its training data consists of online open-source code from GitHub and GitLab (28 languages in total) with the great majority of files being Python and JavaScript.
- *GPT-Neo* [16] GPT-Neo is a GPT-3-style model based on the EleutherAI's replication. It is trained on ThePile dataset [33], a large corpus containing a blend of natural language texts and code from various domains.
- *CodeGen* [65] CodeGen is a standard transformer-based autoregressive model for program synthesis released by Salesforce. CodeGen is trained sequentially on three datasets (i.e., ThePile [33], BigQuery [42], and BigPython [65]), resulting in three corresponding models CodeGen-NL, CodeGen-Multi, and CodeGen-Mono, respectively.
- *PolyCoder* [92] PolyCoder builds on the GPT-2 architecture [69] and is trained on a code dataset of 249 GB, which is collected from GitHub in 12 programming languages.
- *CodeT5* [90] Different from the above encoder-only code generation models, Salesforce's CodeT5 is an encoder-decoder transformer-based model with the same architecture as T5 [70]. It is trained on CodeSearchNet [46] and two collected datasets of C/C# from BigQuery [42].

We note that our goal is to dissect the accountability of PLG models in multiple programming languages. Therefore, we do not include models that are trained solely on code data in Python or Java languages, such as CodeParrot [86], CodeGPT [58], PyCodeGPT [96], PLBART [8], and PYMT5 [26].

[1]https://github.com/wanlunsec/CodeForensic

B. Datasets

We conduct our experiments on the following code generation benchmark datasets: MBPP [13], APPS [40], PY150 [71], and MBXP [12]. These benchmark datasets are commonly involved in prior code-generation-related studies [32], [65], [90], [92].

- *MBPP* [13] The Mostly Basic Programming Problems (MBPP) dataset released by Google includes 974 crowd-sourced programming problems with natural language descriptions in English and corresponding hand-written solutions in Python.
- *APPS* [40] The Automated Programming Progress Standard (APPS) dataset consists of 10,000 coding problems of varying difficulty levels. These coding problems are collected from different open-access coding websites.
- *PY150* [71] This is a line-level code completion dataset processed by Microsoft [58]. It consists of 100,000 Python source code for training and 50,000 for testing.
- *MBXP* [12] Different from the above three Python-specific datasets, Most Basic X(Java/Go/Ruby, etc.) Programming Problems (MBXP) is a collection of multi-lingual datasets. Each sub-dataset (e.g., MBXP-Java) is generated by a conversion framework that translates prompts and test cases from the original Python dataset MBPP to the corresponding data in a target language.

C. Perplexity of Language Models

Generally, the PLG model is also a language model (LM). For a sequence of t discrete tokens $x=\{w_1,\ldots,w_t\}$, LMs can assign it a probability, which measures how confident a language model predicts such a sequence of tokens [14]. The probability $Pr(w_1,\ldots,w_t)$ is computed by applying the chain rule of probability:

$$Pr(w_1,\ldots,w_t) = \prod_{i=1}^{t} Pr(w_i|w_1,\ldots,w_{i-1}) \tag{1}$$

Rather than directly computing (1), a more commonly used metric is $Perplexity$ metric [45], [59]. A lower perplexity indicates higher model confidence, i.e., higher likelihood of the code snippet. The $Perplexity$ metric is computed as:

$$PPL(x) = \exp\left\{-\frac{1}{t}\sum_{i=1}^{t}\log Pr(w_i|w_1,\ldots,w_{i-1})\right\} \quad (2)$$

III. Training Usage Audit

The powerful memorization of PLG models poses a risk of reproducing data directly from the training data, which potentially raises legal concerns over copyright infringement and ownership [24], [39]. In this section, we investigate the legal and ethical issues of current PLG models from the perspective of model development which is to audit the training data usage of PLG models. We also propose to use the membership inference approach to answer *RQ1*: Is a given code snippet used in the PLG model's training data?

A. Intuition and Goals

Membership inference utilizes the difference in the model's behaviors on training data (labeled as "member") and others (labeled as "non-member") [81], as the models are inclined to behave more confidently on member data than on non-member data. In other words, the model's loss on an individual data sample directly reflects how well the model has memorized this data sample. Therefore, we can utilize the membership inference to audit the training usage of the PLG model, where we aim to achieve the following goal:

- *Determining Membership:* The goal is to determine whether a given (human-written) code snippet is used for training the PLG model.

B. Methodology

Following previous works [60], [61], [95], we assume that we only have access to the output sequences and their probabilities from the target model. We have further discussion about this assumption in Section III-E. Model loss-based membership inference methods predict the data sample with a lower loss to be a member of the training data [48], [95]. Although the model's loss can reflect the membership status of each individual target data, recent research [23], [60], [94] show that prior methods that rely solely on the loss of the target model yield sub-optimal performance since the loss value provides a weak indicator for membership prediction. Therefore, we utilize an advanced reference-model-based likelihood ratio test [63], [94]. According to the Neyman-Pearson lemma [64], membership inference based on likelihood ratio test can obtain the highest testing power (i.e., achieve optimal performance) [23]. Specifically, we use the likelihood ratio test to distinguish the following two hypotheses [60], [63]:

- Null hypothesis (H_{out}): The code snippet x is drawn from the general population, independently from the training dataset of the PLG model.
- Alternative hypothesis (H_{in}): The code snippet x is drawn from the training dataset of the target PLG model.

Let $\mathbf{G}_T$ and $\mathbf{G}_R$ be the target and reference PLG model, respectively. The reference model $\mathbf{G}_R$ is trained on data samples drawn from the general population. The likelihoods for H_{out} and H_{in} are denoted as $Pr[x;\mathbf{G}_R]$ and $Pr[x;\mathbf{G}_T]$, respectively. Therefore, the log-likelihood statistic is computed as follows

$$L(x) = \log\left(\frac{Pr[x;\mathbf{G}_R]}{Pr[x;\mathbf{G}_T]}\right) \quad (3)$$

Here, we calculate the log-likelihood statistic using the perplexity value:

$$L(x) = \log\left(\frac{PPL^{-1}(x;\mathbf{G}_R)}{PPL^{-1}(x;\mathbf{G}_T)}\right) \quad (4)$$

If the log-likelihood ratio statistic $L(x)$ is smaller than threshold ϵ, we can reject H_{out} and classify it as a training set member. Otherwise, we classify it as a non-member. In order to determine the threshold ϵ, we first empirically estimate the null distribution of the test statistic $L(x)$ by calculating $L(x)$ for all x drawn from the general population distribution. We then select the threshold ϵ such that the false positive rate (over members and non-members) would not exceed a pre-defined level α.

The quality of a test can be measured by the test power (i.e., the true positive rate) under a pre-defined error rate (α), i.e., the false positive rate. Higher test power for a lower error rate indicates a better ability to distinguish between members and non-members. The test power under different thresholds can be characterized by the Receiver Operating Characteristic (ROC) curve. Therefore, following previous works [23], [60], [94], we use the AUC (Area Under the Curve) score to evaluate all membership inference methods.

C. Experiment Setup

Dataset Configurations. We use the APPS dataset which consists of 5000 training samples and 5000 testing samples. To broaden our evaluation on a larger-scale dataset, we conduct evaluation on the PY150 dataset [71]. We fine-tune the pre-trained models on the training samples of each dataset using Adam optimizer with a learning rate of 10^{-5}. Consequently, testing samples are regarded as non-members. For further experiment setup details, please refer to Appendix B.1, available online.

Model Configurations: We use pre-trained but not fine-tuned models as the domain-specific reference models, while the fine-tuned models are treated as target models. Here, we study the training usage auditing on CodeGen, GPT-Neo, and CodeGPT [58]. We include CodeGPT because it shares the same architecture as GPT-2, allowing us to use the standard pre-trained GPT-2 as a general-domain reference model for conducting an ablation study.

D. Results & Analysis

This section presents the main results of the membership inference. Table II shows the performance of different membership inference approaches, including target model loss-based method (LOSS-Based) [48], likelihood ratio test using a general-domain reference model (LRT-G), and likelihood ratio test using a domain-specific reference mode (LRT-S). We can observe that

TABLE II
AUC Scores of Different Membership Inference Approaches, Including Model Loss-Based (LOSS-Based), Likelihood Ratio Test Using a General-Domain Reference Model (LRT-G), and the One Using a Domain-Specific Model (LRT-S)

Dataset ↓	Target Model →	CodeGen	GPT-Neo	CodeGPT
APPS	LOSS-Based	56.2	61.9	52.8
	LRT-G	-	-	67.7
	LRT-S	**79.4**	**89.7**	**71.1**
PY150	LOSS-Based	72.2	64.9	56.9
	LRT-G	-	-	72.9
	LRT-S	**80.7**	**74.9**	**75.6**

LRT-based methods (LRT-S and LRT-G) significantly outperform the LOSS-based one. For example, the AUC scores of LRT-S and LRT-G are at least 10% higher than those of the LOSS-Based method on all datasets and target models. This indicates that LRT methods can better differentiate similar samples than loss-based methods because the reference model used in LRT methods can magnify the difference of the target model's behavior between members and non-members.

In addition, we find that it is better to use a code-domain reference model when using the LRT-based membership inference to answer RQ1. As we can see in Table II, LRT-G using the GPT-2 model achieves comparable performance, but is worse than LRT-S using the pre-trained CodeGPT model. This is mostly due to the overlap in domains between the reference model (pre-trained CodeGPT) and the target model (fine-tuned CodeGPT). In contrast, GPT-2 is trained on the natural language domain. This indicates that an ideal reference model for this approach would be trained on a similar data distribution to that of the target model in order to better characterize the data memorization of the target model.

E. Discussion

Both the LOSS-based method and LRT-based method are considered as black-box membership inference in previous research [48], [94], where they assume only access to the probability value from the target model but not to its parameters or internal computations. However, it is not practical to obtain probability value from the PLG-model-based services, since users can only obtain the predicted results (generated code) from such black-box services. Although existing shadow-model-based membership inference can perform audits using only the generated content, they either yield poor performance on large transformer-based language models [41] or merely focus on shallow and simple LSTM models [82].

To ensure regulatory compliance of PLG models with respect to training data usage, we recommend concerted efforts from both technical and regulatory perspectives. First, there is a pressing need to carry out more research on developing robust membership inference methods that can effectively audit PLG models using only generated content from PLG models. Second, given the limitations of current membership inference methods, regulators can mandate PLG model developers to provide internal access to their models to allow for a reliable audit of the training data usage. Third, model developers can be required to publish or provide clear documentation on the training data used for PLG models, including the sources and legal status of the data. These actions will enhance transparency and accountability in the development and deployment of PLG models, and help mitigate potential legal and ethical risks.

- Given access to the output probabilities, the LRT-based membership inference method can be used to audit training data usage of PLG models.
- Limitations of current membership inference methods in practical application emphasize the need for concerted efforts from technical and regulatory perspectives to enhance accountability in the development of PLG models.

IV. Neural Code Detection

Aside from the accountability of model development, the potential misuse of PLG models motivates us to investigate their accountability in terms of model deployment. To this end, our study of model deployment accountability starts with Neural Code Detection (*RQ2*) that questions how to distinguish between neural code and human-written code. To answer this question, we propose a learning-based method that builds a detector to classify a code snippet as human-written or model-generated. Specifically, this section begins with our intuitions and goals, followed by the pipeline of building the detector, and finally presents the experimental setup and the evaluation results.

A. Intuitions and Goals

Our intuition behind the neural code detection comes from research in natural language generation (NLG) models [47], [72], [87], [97], and specifically driven by their conclusion that different NLG models share common or similar fingerprints. We infer that PLG models have similar behaviors in this aspect. Namely, PLG models share common or similar fingerprints, and these fingerprints can be leveraged to detect neural code from human-written code. Therefore, our learning-based method for neural code detection aims to achieve the following two goals:

- *Detect neural code generated by known models:* The primary goal for detection is to distinguish neural code from human-written code where the neural code for training and testing is generated by the same PLG model.
- *Detect neural code generated by unseen models:* Due to the growing proliferation of PLG models, it is impossible to include every PLG model when learning the detection model. Therefore, the detection model should be able to detect neural code generated by unseen models. This can verify that the common fingerprints captured by the detector generally exist in all PLG models rather than in some specific models.

B. Methodology

To achieve the two aforementioned goals, we propose to learn a detector to capture the common fingerprints among PLG models. The detector is intrinsically a function mapping any code snippet to a label, i.e., neural or human-written. Therefore, the training data should consist of neural and human-written code snippets. The human-written code can be directly obtained from the benchmark dataset (e.g., MBPP) while the neural code is generated by PLG models. More concretely, the learning pipeline contains the following four stages:

Stage 1: Generate Neural Code. The neural code is generated by inputting prompts to PLG models. Here we uniformly notate code snippets as $\mathcal{X}_k := \{x_i^k\}_{i=1}^N$, where $k = 0$ means human-written code and $k > 0$ means neural code. To evaluate the generalization of the learned detector, we here only use one PLG model, i.e., $\mathbf{G}_k$ from a set of PLG candidates $\mathcal{G} := \{\mathbf{G}_k\}_{k=1}^K$ to generate neural code in the training set and observe its detection effectiveness.

Stage 2: Generate Labels. The second stage for training set preparation is to label the snippets of code. For each code snippet x_i^k, its label y_i will be 0 if $k = 0$ while 1 if $k > 0$. Given paired x_i^k and y_i, our training set is denoted as $\mathcal{D} := \{(x_i^k, y_i) | i = 1, \ldots, N, k = 0 \text{or}\, k \in \{1, 2, \ldots, K\}\}$. Given that we exclusively include neural code from a single PLG model in constructing the training dataset, the dataset maintains an equal count between neural code and human-written code snippets.

Stage 3: Learn the Detector. With the training set $\mathcal{D}$ at hand, we can train a binary classifier $\mathcal{F}$ mapping a code snippet to a probabilistic vector $\mathcal{F}(x_i)$ over all possible classes. Training $\mathcal{F}$ is to solve the following optimization problem: $\min_{\mathcal{F}} \Sigma_{x_i \in \mathcal{X}} L(\mathcal{F}(x_i), y_i)$, where $L(\mathcal{F}(x_i), y_i)$ is a loss function that measures the differences between the model output $\mathcal{F}(x_i)$ and the ground truth label y_i.

Stage 4: Inference. After the training stage, given a code snippet x, the label of x is $\operatorname{argmax} \mathcal{F}(x)$. In other words, we consider the value with the highest probability as the most possible class $\hat{y}$ for the source-unknown code snippet x. We note that our testing neural code snippets are generated not only by model $\mathbf{G}_k$ but also by models in $\mathcal{G}/\mathbf{G}_k$.

C. Experimental Setup

Dataset Configurations: In this experiment, we leverage two datasets: the MBPP and APPS datasets. The MBXP dataset is excluded due to the absence of human-written code solutions. Unless stated otherwise, we only use the test set of the APPS dataset, which contains 5,000 prompts and corresponding solutions. Our experimental results show that a size of around 500 prompts is sufficient for training a well-performed detector. All prompts from each dataset are employed to generate neural code snippets. Subsequently, 80% of these snippets, chosen randomly, are paired with their corresponding human-written solutions for training, while the remaining 20% of both neural and human-written code snippets are held for evaluation.

Code Generation Models and Configurations: We choose the larger-size version of each code generation model by default, i.e., CodeGen-6B, InCoder-6B, PolyCoder-2.7B, GPT-Neo-2.7B, and CodeT5-base (220 M). We choose CodeGen-6B instead of CodeGen-16B for the sake of generation efficiency. For the sampling parameters for all PLG models, we use the

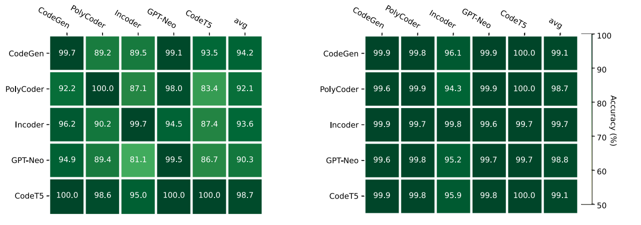


Fig. 2. Accuracy of neural code detection on (a) MBPP and (b) APPS datasets. The PLG models on the y-axis are used to generate neural code for the training set, while models on the x-axis are used to generate neural code for evaluation (same for the figures in the following sections).

softmax(x/T) sampling with a small temperature parameter $T = 0.2$, since a lower temperature (more greedy decoding) achieves better generation quality when only generating one code snippet [13], [25], [32], [51], [65], [92]. Additionally, we use the most commonly used top-p sampling (so called nucleus sampling) [43] with $p = 0.95$ [25], [32], [65], [90], [92].

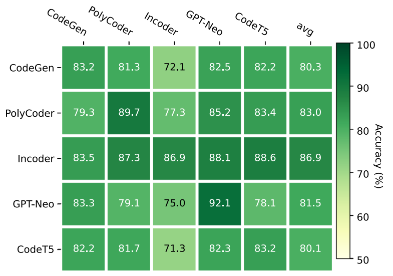
Detector Construction: When building our detector, we fine-tune UniXcoder [36], a state-of-the-art pre-trained programming language model, to improve the effectiveness of neural code detection. UniXcoder employs cross-modal information, including Abstract Syntax Tree (AST) structures and code comments, to enrich code representations. Pre-trained on the CodeSearchNet dataset and an additional dataset featuring C, C++, and C# programming languages, UniXcoder can support for nine programming languages: Java, Ruby, Python, PHP, JavaScript, Go, C, C++, and C#. We use AdamW optimizer [57] with learning rate $r = 10^{-6}$.

Metric: In line with prior studies [47], [72], [97], we evaluate the performance of the neural code detector in terms of classification accuracy with balanced positive and negative samples.

D. Results & Analysis

This section presents the main results of the detection performance. Fig. 2 shows the performance of our neural code detector in *in-domain* and *cross-domain* settings. For the in-domain setting where training and testing neural code snippets are generated by the same model (i.e., results on diagonal entries), our detector achieves almost 100% accuracy on both datasets. This indicates the existence of fingerprints in each PLG model, and such fingerprints are inherited by their generated code. Moreover, the results show that our detector can leverage these fingerprints to detect neural code generated by known models, achieving the first goal.

For the cross-domain setting where training and testing code snippets are generated by different generation models (i.e., results on off-diagonal entries), our detector achieves average accuracy of >90% and >98% on both datasets, respectively. These cross-domain results indicate the existence of common fingerprints shared across PLG models. This allows us to effectively detect neural code, achieving the second goal of detecting neural code generated by unseen models. Furthermore, we evaluate the performance of our detector in the case that training



(a)

Fig. 3. Accuracy of neural code detection when training and testing neural code snippets are generated with prompts from MBPP and APPS datasets, respectively.

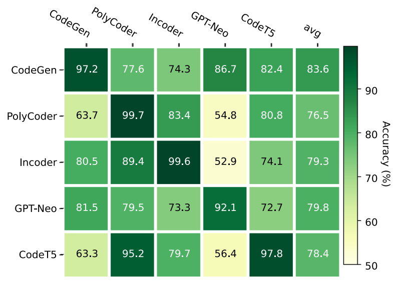


Fig. 4. Accuracy of neural code detection on the MBPP dataset when using a well-performed neural text detector.

and testing prompts are from different datasets. The results are shown in Fig. 3. We can observe that our detector achieves $>80\%$ average accuracy when trained with MBPP and tested on APPS. These results indicate that our neural code detector has good generalization ability.

We can also observe that our detector yields better performance on APPS. The reason is that APPS has a larger size of the training data than the others. Specifically, APPS has 4,000 neural code snippets and another 4,000 corresponding human-written code snippets for training. In contrast, MBPP only has $779 \times 2 = 1558$ snippets. Additionally, our additional experiments find that the performance of the detector will not drop significantly on MBPP and APPS when we manually decrease the size to at least $500 \times 2 = 1000$. Therefore, we recommend building the detector with as least 1,000 snippets to achieve better detection performance.

Can Neural Text detector Handle Neural Code? The previous work [47] showed that their fine-tuned BERT-based classifier could outperform other methods in detecting neural texts. We adopted this well-performed neural text detector to detect neural code snippets generated by each PLG model. Specifically, we keep the same setting as previous experiments and just alter the detector to the neural text detector in the inference stage for evaluation. We use the same testing split of neural code snippets generated by each PLG model using prompts from the MBPP dataset. The results show that it yields accuracy ranging from 52.1% to 65.8%, indicating the inability of the neural text detector to handle neural code.

Furthermore, we evaluate the detection performance when fine-tuning BERT [28] using the training data $\mathcal{D}$ constructed from the MBPP dataset. As shown in Fig. 4, we can find that this BERT-base detector achieves almost the same performance as our UniXcoder-based detector (Fig. 2(a)) in the in-domain setting. However, it shows significant performance decreases in generalization ability. For example, the accuracy values of the BERT-base detector are all greater than 92% for the in-domain setting, while it drops to nearly 50% when the testing code snippets are generated by GPT-Neo but the training data is generated by PolyCoder, InCoder, or CodeT5. This indicates that a purely natural language-based pre-trained model, such as BERT, lacks the ability to understand the distinct semantic and syntactic information of programming languages. These results highlight the importance of exploring the properties of PLG models for neural code attribution.

- Extensive evaluations demonstrate the existence of common fingerprints shared across different PLG models.
- The inefficiency of the neural text detector highlights the critical need to investigate the features of PLG models and their generated neural code.

V. Challenges of Neural Code Attribution

Section IV shows that we can effectively distinguish neural code from human-written code by learning a binary classifier. However, this can only tell us the given neural code snippet is generated by *some* models. In this section, we delve deeper to investigate the problem of neural code attribution (*RQ3*), i.e., attributing a given neural code snippet to *the* model (e.g., a suspected model). Unfortunately, our empirical analysis and theoretical understanding demonstrate that it is NOT feasible to address this one-to-one attribution problem due to inherent challenges in neural code attribution.

A. Empirical Analysis

In this section, we propose two intuitive methods that attempt to solve the problem of neural code attribution. Our experimental results demonstrate the ineffectiveness of these two methods.

1) One-Class Classification: One may intuitively formulate the one-to-one attribution problem (where only one model is known) as a one-class classification problem (where only one class is known). Therefore, we can build a one-class classifier with neural code generated by the known model to detect neural code generated by unknown models. To examine whether the one-class classifier can effectively address this problem, we employ the widely used one-class support vector machine (SVM) [77] with a Gaussian kernel.

The preparation of the training set is similar to Stage 1 described in Section IV-B. The difference is that we do not need to use human-written code here. As we cannot fine-tune UniXcoder in the one-class classification problem, we use it as a feature extractor to transform the raw code data into numerical features that can be processed by the one-class SVM. In this

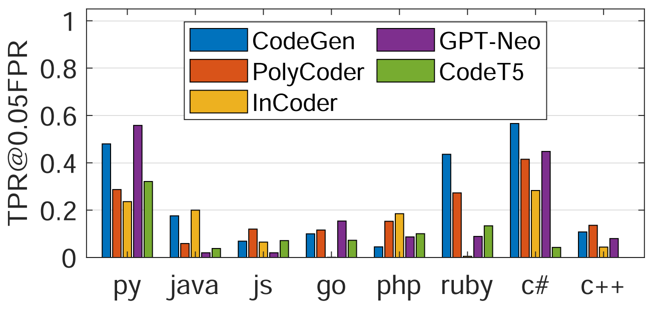


Fig. 5. Performance of attribution model using one-class SVM on different programming languages.

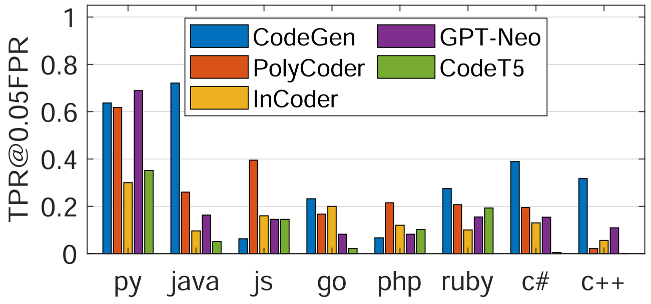


Fig. 6. Performance of attribution model using likelihood-based attribution on different programming languages.

way, our one-class SVM model can also benefit from its code understanding ability. Following the common practice [28], [36], we take the last hidden state at the final layer of UniXcoder as the embedding feature of code x.

We evaluate the performance of one-class SVM detectors for each PLG model on all programming language datasets except for C++ language because C++ is not supported by CodeT5. The results are reported in Fig. 5. It can be observed that the performance of using one-class SVM for neural code attribution is very poor in most cases. For example, the one-class SVM detector trained with code snippets generated by CodeGen yields true positive rates (TPRs) of only 6.9%, 10%, and 4.5% on Java, Go, and PHP, respectively with a fixed false positive rate (FPR) at 5%. Similar to the results for CodeGen, the average TPR @ 5% FPR on all datasets for Polycoder, InCoder, GPT-Neo, and CodeT5 are only 19.5%, 12.7%, 18.2%, and 11.14%, respectively.

To better understand the failure of the one-class classification method, we also analyze the distributions of Python code snippets generated by different PLG models. Like the one-class classification method described above, we first use the pre-trained UniXcoder as a feature extractor to transform the raw code snippets into feature vectors. Then, we employ t-SNE [88] to project these features into 2D space. The t-SNE method is an unsupervised dimensionality reduction technique that visualizes high-dimensional data by mapping points into a lower-dimensional space, which is effective at understanding the underlying structure or relationships between data points. Fig. 7 illustrates the representation distributions of Python code

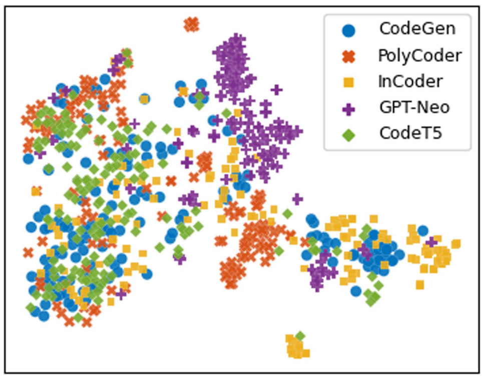


Fig. 7. Representation distributions of Python code generated by different PLG models. The representations are extracted by UniXcoder and projected by t-SNE [88].

snippets generated by different PLG models. We can observe that the output distributions of different PLG models are heavily overlapped, impeding the one-class SVM model from learning a clear support boundary.

2) Likelihood-Based Attribution: The one-to-one attribution problem may also be addressed by comparing the likelihood that a code snippet is generated by a given PLG model, as the PLG model would give a higher likelihood to its own generated code snippets compared to the code snippets generated by other models. Similar to the likelihood ratio test in Section III-B, we use the perplexity value of each sample to perform our likelihood-based attribution. Specifically, for a given PLG model $\mathbf{G}$, we compute the perplexity value $PPL(x)$ for each code snippet x using (2). Subsequently, the likelihood-based attribution method classifies a code snippet as positive (i.e., generated by the model $\mathbf{G}$) if its perplexity value falls below a predefined threshold. Otherwise, we mark it as negative (i.e., not generated by the model $\mathbf{G}$). This threshold is determined based on a fixed false positive rate.

We report the results of the likelihood-based attribution on different programming language datasets in Fig. 6. We can observe that likelihood-based attribution cannot effectively determine whether a code snippet is generated by the given model. For example, although likelihood-based attribution achieves 72.1% TPR @ 5% FPR when distinguishing Java code snippets generated by CodeGen from those generated by the other models, it yields only 6.3%, 23.2%, and 6.7% on Javascript, Go, and PHP, respectively. In most cases, the TPRs @ 5% FPR of likelihood-based attribution is around 20%. The average TPR @ 5% FPR on all datasets for Polycoder, InCoder, GPT-Neo, and CodeT5 is only 25.9%, 14.5%, 19.7%, and 12.4%, respectively.

B. Understanding the Failure

The empirical analysis in Section V-A has demonstrated the failure of using two intuitive methods to solve the problem of neural code attribution. In this section, we provide a theoretical analysis to prove that it is an impossible task to reliably attribute a single code snippet to a given PLG model. We formalize our *RQ3* as a single-instance goodness-of-fit hypothesis test [79], [91]. Subsequently, we prove that *RQ3* as a single-instance distributional test is impossible, i.e., the test does no better than random guessing. This is done by generalizing theoretical results from previous works in OOD detection [99] to our neural code attribution setting.

For a given code snippet x and a PLG model $\mathbf{G}$, the goal of *RQ3* is to determine whether x comes from the model $\mathbf{G}$'s output distribution $\mathbb{P}$ or not. This can be formally defined as follows:

Definition V.1. (Single-Instance Goodness-of-Fit Test) Formally, given an instance x, the test decides whether to reject the null hypothesis (H_0) that x was drawn from the data distribution $\mathbb{P}$, in favor of an alternative hypothesis (H_1) that x came from a distribution other than $\mathbb{P}$:

$$H_0 : x \sim \mathbb{P}, \tag{5}$$

$$H_1 : x \sim \mathbb{Q} \in \mathcal{Q}, \mathbb{P} \notin \mathcal{Q}, \tag{6}$$

where $\mathcal{Q}$ is a set of alternative distributions, i.e., the unknown output distributions of other PLG models except for the given model $\mathbf{G}$.

Let $\phi(x)$ be a predetermined test statistic, which is a function of a single instance x. Based on the value of $\phi(x)$, one can decide whether to reject the null hypothesis or not. The quality of a test is measured by the test power (i.e., TPR) under a pre-defined significance level (α), also known as the Type I error or false positive rate.

The formalized single-instance goodness-of-fit test is a challenging statistical inference task given that there is no knowledge about the output distributions of other PLG models. As shown in Proposition 1 in the previous work [99], the single-instance distributional test formalized in Definition V.1 has test power equal to the false positive rate, which is equivalent to random guessing. Proposition 1 in [99] states that no test statistic can do well against all alternative data distributions. In other words, any test statistic $\phi(x)$ for single-instance distributional testing must trade off their test power against different unknown alternative distributions $\mathbb{Q} \in \mathcal{Q}$.

For further explanation, we use an intuitive example to elaborate on the impossibility of the single-instance distributional test. As aforementioned, we consider test statistics that directly utilize knowledge of the output distribution $\mathbb{P}$, which can be denoted as $\phi_{\mathbb{P}} : \mathcal{X} \to \mathbb{R}$. Suppose $\mathcal{X} \in \mathbb{R}^d$, then the test statistic $\phi_{\mathbb{P}}$ is a mapping function from $\mathbb{R}^d \to \mathbb{R}$. Therefore, the statistic value $\phi_{\mathbb{P}}(x)$ is a projection of d-dimensional distribution onto a one-dimensional space of the test statistic. Since a single marginal distribution cannot capture all differences between two multivariate distributions, the projections of $\mathbb{P}$ and $\mathbb{Q}$ on the test statistic cannot assess all their differences.

The above theoretical analysis suggests that, without additional knowledge about the output distributions of other PLG models, no method for one-to-one attribution can guarantee performance beyond random chance. This indicates one-to-one attribution is an impossible task. Therefore, we propose two feasible alternative questions by relaxing the constraints on the number of candidate models and the number of neural code snippets, respectively. Specifically, we present to address *attribution classification* (*RQ3.1*) which aims to classify a given neural code snippet to its source model among a set of candidate models (Section VI), and *attribution verification* (*RQ3.2*) which aims to verify whether a set of neural code snippets is generated by a given PLG model (Section VII).

- Our empirical and theoretical analyses demonstrate that it is impossible to reliably attribute the generation of one code snippet to a specific model, shedding light on the challenges of dissecting neural code accountability.

VI. Attribution Classification

In this section, we present our study of attribution classification, addressing the *RQ3.1*, i.e., given a neural code x and K candidate PLG models, can we single out the source model (among K alternatives) that generated x? To answer this question, we learn a multi-class classifier to predict which PLG model generates the neural code snippet. We begin with our intuition and goals. We then detail our method to build the multi-class classifier. Finally, we present the experimental setup and evaluation results.

A. Intuition and Goals

We drive our intuition behind the attribution classification from the fact in NLG models that neural texts generated by NLG models carry subtle but discriminating features or fingerprints inherited from their source models and that these features can be utilized by machine learning (ML) algorithms for attribution [62], [68], [87]. Therefore, we believe that PLG models also hold the same properties, i.e., code snippets generated by PLG models inherit subtle but different fingerprints from their source PLG models. To this end, we perform an attribution classification approach aiming to achieve the following goals:

- *Identifying Source Models:* The primary goal of attribution classification is to predict which PLG model generates the neural code among a set of model candidates. This can verify whether the generated code carries distinguishing fingerprints inherited from their source models.
- *Generalization to different prompts:* In real-world settings, neural code snippets can be generated by prompts from different datasets. As a result, the distribution of the prompts may also be different. Therefore, the attribution classifier should have good generalization performance when neural code snippets for training and testing are generated by prompts from different datasets.

B. Methodology

We propose a multi-class classification learning method to achieve the above-mentioned two goals for attribution classification. Similar to neural code detection, the pipeline consists of four stages.

Stage 1: Generate Neural Code. We use a set of prompts to query each model in $\mathcal{G} := \{\mathbf{G}_k\}_{k=1}^{K}$ to generate a set of neural code $\mathcal{X}_k := \{x_i^k\}_{i=1}^{N}$.

TABLE III
PERFORMANCE COMPARISON OF ATTRIBUTION CLASSIFICATION ACROSS THE DIFFERENT PROGRAMMING LANGUAGES IN TERMS OF CLASSIFICATION ACCURACY

	py	**java**	**js**	**go**	**php**	**ruby**	**c#**	**c++**	**all***
UniXcoder-based	**0.994**	**0.955**	**0.900**	**0.872**	**0.941**	**0.970**	**0.893**	**0.915**	**0.932**
CodeBERT-based	0.990	0.889	0.873	0.751	0.905	0.966	0.789	0.850	0.891
BERT-based	0.914	0.882	0.826	0.719	0.882	0.953	0.775	0.829	0.869
DL-CAIS [6]	0.711	0.844	0.720	0.690	0.769	0.863	0.742	0.786	-
PbNN [18]	0.702	0.861	-	-	-	-	-	0.750	-
PbRF [18]	0.812	0.906	-	-	-	-	-	0.877	-

* the attribution classifier is trained and evaluated in the multi-lingual setting.

Stage 2: Generate Label. For each x_i^k in $\mathcal{X}_k$, we assign $y_i = k$ as its label. We construct our training set as $\mathcal{D} := \{(x_i^k, y_i) | i = 1, \ldots, N, k = 1, \ldots, K\}$.

Stage 3: Learn the Multi-class Classifier. We train a multi-class classifier $\mathcal{F}$ with the constructed dataset $\mathcal{D}$ for addressing the attribution classification problem.

Stage 4: Inference. After the training stage, we can evaluate the learned multi-class classifier to predict the most possible source model of a given testing code snippet x, which may be generated by any model $\mathbf{G}_k$ in $\mathcal{G}$.

C. Experimental Setup

Since our attribution classification task is very similar to neural code detection in Section IV, we mainly describe the different parts of the experimental setup for attribution classification.

Dataset Configurations: We use MBPP and MBXP to evaluate the performance of our attribution classifier for multiple programming languages, i.e., MBPP for Python, MBXP for Java, Javascript, Go, PHP, Ruby, C#, and C++. Similar to neural code detection, we randomly split the prompts with an 80–20 ratio to generate neural code snippets for training and testing, respectively.

Code Generation Models: As shown in Table I, different types of PLG models are trained on different sets of programming languages. Therefore, we do not include CodeT5 for MBXP-CPP dataset since the training dataset of CodeT5 does not contain any code in the C++ program language.

Classifier Construction: When building our attribution classifier, we fine-tune UniXcoder [36] using the constructed dataset. We use the same AdamW optimizer [57] with a learning rate $r = 10^{-5}$.

Metric: As described in Section VI-B, we construct a multi-class classifier for attribution classification. The number of output classes in the classifier corresponds to the number of available PLG models. It is important to note that this classifier addresses a multi-class classification task, with a balanced distribution of test samples across each class. In line with prior studies [6], [18], [62], [68], we adopt classification accuracy as our evaluation metric.

D. Results & Analysis

We now present the main results of the attribution classification. Table III shows that our attribution classification obtains good performance, achieving the first goal. For instance, in

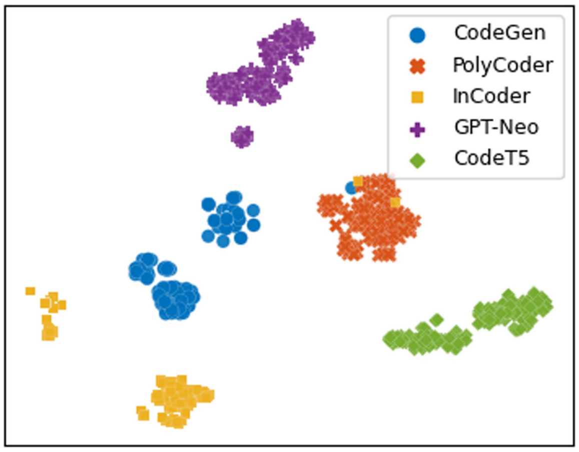


Fig. 8. Representation distributions of Python code generated by different PLG models. The representations are extracted by the fine-tuned UniXcoder in attribution classification and then projected by t-SNE [88].

the mono-lingual attribution classification setting where our attribution classifier is trained and tested on mono-lingual neural code snippets, our attribution classifier achieves accuracy scores from 87.2% to 99.4% in different programming languages. In the multi-lingual setting where our attribution classifier is trained and tested on neural code snippets in all languages, our attribution classifier still maintains a remarkable performance, achieving an accuracy score of 93.4%. These results indicate that code snippets generated by PLG models inherit subtle but unique fingerprints from their source PLG models, leading to the success of our attribution classifier in identifying the fingerprints.

Furthermore, we evaluate our attribution classifier's generalization ability in the case that training and testing prompts are from different datasets. We find that our classifier achieves an accuracy score of 92.1% when training with APPS and testing on MBPP. When we compared it to the classifier accuracy of both training and testing on MBPP, i.e., the 99.4% of Python result in Table III, 92.1% is still a comparably satisfied accuracy. This indicates that our attribution classifier has a good generalization ability to neural code snippets that are generated by prompts from different distributions, achieving the second goal.

To illustrate the efficacy of our proposed attribution classification approach, we visualize the representation distributions of Python code snippets generated by various PLG models in Fig. 8. Notably, by relaxing constraints on the number of available PLG models, we can fine-tune the pre-trained UniXcoder model to construct our attribution classifier using code generated from these diverse models. In contrast, the one-class classification relies solely on the pre-trained UniXcoder model as a feature extractor. Consequently, compared to the considerable overlap observed in Fig. 7, the representation distributions of Python code from different PLG models in Fig. 8 exhibit distinct separation, contributing to the high accuracy achieved by our attribution classifier. These findings align with our theoretical analysis, emphasizing the necessity of incorporating additional knowledge regarding the output distributions of diverse PLG models for robust neural code attribution.

Comparison With Authorship Attribution of Human-Written Code: The task of our attribution classification for neural code is very similar to a well-studied area, source code authorship attribution [6], [18], which aims to identify the author of a given piece of code from a predefined set of authors. Here, we compare our attribution classification method with three human-written source code authorship attribution baselines, including DL-CAIS [6], PbNN [18], and PbRF [18]. These methods are the most recent and state-of-the-art ones, and they are open-sourced. Since PbRF and PbNN require a specific parser for each target language, their implementation only supports four programming languages, including Python, Java, C, and C++.

As shown in Table III, we can observe that our proposed attribution classification method (UniXcoder-based one) outperforms all three human-written source code authorship attribution methods. For example, our proposed method achieves an accuracy score of 99.4% in Python while DL-CAIS, PbNN, and PbRF yield only 77.1%, 70.2, and 81.2%, respectively. The premise for the success of authorship attribution is that each author has a unique coding style, such as lexical, syntactic, and semantic patterns [54]. Our experimental results show that these authorship attribution methods do not work effectively for neural code attribution. We conjecture that this is because PLG models are trained on large corpora from different authors thus there may be no clear-cut stylometric differences in neural code generated by PLG models.

Impact of Pre-Trained Code Understanding Model: The performance of our attribution classifier relies on the code understanding ability of the pre-trained programming language model. Here, we explore the impact of the choice of code understanding models on attribution performance. The results are shown in Table III. Similar to our findings in Section IV-D, we can observe that the attribution classifier built on a purely natural language-based pre-trained model, like BERT [28], yields the worst performance compared to the classifiers built on pre-trained models for programming languages.

Additionally, UniXcoder is preferable given both UniXcoder and CodeBERT are specifically pre-trained on source code. This is because, in addition to semantic information of code, UniXcoder can leverage syntax information of abstract syntax tree (AST) to enhance code representation learning. Moreover, we should point out that the CodeBERT-based attribution classifier yields comparable performance in six programming languages, i.e., Python, Java, Javascript, Go, Php, and Ruby. However, there are large accuracy drops in the other two programming languages, i.e., C# and C++. We suppose the reason is that CodeBERT is trained on the CodeSearchNet dataset that only contains the first six programming languages. As a result, it does not have a good understanding ability of other languages, such as C# and C++, leading to a significant accuracy drop compared to our proposed UniXcoder-based classifier.

Distinguishing PLG Models in a Family: In the above evaluation, our attribution classifier aims to learn and distinguish fingerprints from different PLG families, where they differ from each other in many ways. For example, as for training objectives, CodeGen, PolyCoder, and GPT-Neo utilize a left-to-right (causal) autoregressive language modeling objective [21], [25], InCoder adopts a causal masking objective [7], and CodeT5 includes masked span prediction [70] and denoising sequence reconstruction [52] objectives. In terms of model architectures, CodeGen, PolyCoder, GPT-Neo, and InCoder are decoder-only models while CodeT5 is an encoder-decoder model.

Here, we explore whether our attribution classification can distinguish fingerprints of more similar models in the same family, i.e., models that are different in model size but designed by the same algorithm. In the experiments, we use models in the CodeGen family as it provides models trained in four sizes, 350 M, 2.7B, 6B, and 16B. Particularly, we conduct this experiment on the MBPP dataset with CodeGen-Mono-350 M, CodeGen-Mono-2.7B, CodeGen-Mono-6B, and CodeGen-Mono-16B. The result shows that our attribution classifier achieves a good accuracy score of 83.87%, much higher than the four-class random guess accuracy of 25%. This result demonstrates that our attribution classification can effectively learn and distinguish fingerprints of similar PLG models that are designed by the same algorithm.

E. Discussion

In the previous evaluations, we demonstrate the subtle but distinctive fingerprints in PLG models, which are effectively captured by our attribution classifier. Here, we aim to visualize the fingerprints of different PLG models.

Similar to source code authorship attribution [6], [18], [54], we can treat each of our PLG models as a code author. We suppose that neural code snippets generated by different PLG models may have different coding style patterns which reflect the unique fingerprints of PLG models. To verify it, we exploit the model interpretability tool, Captum [50] which provides state-of-the-art methods (e.g., Integrated Gradients [84]) to explore which coding style patterns are contributing to the output of the attribution classifier.

We query each model to generate a code snippet from the same prompt that is randomly selected from MBPP. We then use Captum to compute the importance score (ranging from -1 to 1) of each token. If a token has a large importance score, this token positively contributes to the classification. For example, as shown in Fig. C2 in Appendix C, available online, the token "print" is denoted negatively contributing to the classification. This is a reasonable result because "print" is broadly used in most PLG models and cannot be used as a feature to distinguish PLG models. In contrast, the "Parameters" and "Arguments" are considered as positive tokens. This is because although they all mean defining the parameters, PolyCoder uses "Parameters" while InCoder uses "Arguments". Therefore, we can use this feature to help distinguish PolyCoder and InCoder. In summary, our case study finds that when distinguishing neural code snippets generated by different PLG models, the classification model relies on some coding style patterns such as function names, variable names, and local comment format in a function.

- By relaxing the constraints on the number of candidate models, we propose attribution classification to predict the source model of a neural code with high accuracy.
- Extensive evaluations demonstrate different PLG models have distinctive fingerprints which are inherited by their corresponding generated neural code.
- We analyze the distinct fingerprints of different PLG models from the perspective of coding style using the model interpretability tool.

VII. Attribution Verification

In this section, we present our study of attribution verification, addressing the *RQ3.2*, i.e., can we verify whether a set of neural code snippets is generated by a given PLG model? To answer this question, we formulate attribution verification as a two-sample testing problem. We begin with our intuition and goals. We then detail our method based on two-sample hypothesis testing. Finally, we present the experimental setup and evaluation results.

A. Intuition and Goals

Our intuition behind the attribution verification is that the output distributions of different PLG models are different, otherwise, we cannot learn an effective attribution classifier. However, these output distributions are heavily overlapped, leading to the failure of the single-instance attribution as is discussed in Section V-B. Therefore, we here relax the constraint of one testing code snippet and propose to perform a hypothesis test to verify whether the distribution of the testing code snippets is the same as that of neural code snippets generated by the given model. If yes, the neural code snippets are generated by the given model. We aim to achieve the following goals in attribution verification:

- *Verifying Attribution:* The primary goal is to verify whether a set of neural code snippets is generated by the given PLG model.
- *Generalization to different datasets:* The attribution verification method should have good generalization ability to neural code snippets that are generated by prompts from different distributions.

B. Methodology

Suppose we have a PLG model $\mathbf{G}$ and observe a collection of code snippets $\mathcal{X}' := \{x'_i\}_{i=1}^n$ that are i.i.d. instances from some distribution $\{x'_i\}_{i=1}^n \sim p(x'|\mathbf{G}')$, where $\mathbf{G}'$ is an unknown PLG model and n denotes the number of code snippets. The goal is to verify whether $\mathcal{X}'$ was drawn from $p(x|\mathbf{G})$.

To achieve that goal, we use two-sample testing techniques [35], [55], [56]. More specifically, we have a set of testing instances (i.e., neural code snippets) $\{x'_i\}_{i=1}^n$ to be verified, and we can generate a set of neural code snippets from the model $\mathbf{G}$, $\{x_i\}_{i=1}^m \sim p(x|\mathbf{G})$. Then we can test whether these two sets of code data are drawn from the same distribution (i.e., generated by the same model) or not.

Now we can formulate our hypothesis test as a two-sample testing problem. Suppose we have instances $\mathcal{X} := \{x_i\}_{i=1}^m$ and $\mathcal{X}' := \{x'_i\}_{i=1}^n$ drawn i.i.d. from distributions $\mathbb{P}$ and $\mathbb{Q}$, respectively. The two-sample testing assesses whether distributions $\mathbb{P}$ and $\mathbb{Q}$ are different based on the instances drawn from each of them.

A well-known test statistic for addressing this problem is the Maximum Mean Discrepancy (MMD) [35]:

$$\mathrm{MMD}^2(\mathbb{P},\mathbb{Q}) = \sup_{f\in\mathcal{H}} |\mathbb{E}_{x\sim\mathbb{P}}[f(x)] - \mathbb{E}_{x'\sim\mathbb{Q}}[f(y)]| \quad (7)$$

$$= \|\mu_{\mathbb{P}} - \mu_{\mathbb{Q}}\|_{\mathcal{H}}^2, \quad (8)$$

where f is a function of the reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ associated with a positive definite kernel k. In practice, a popular option is to use the Gaussian kernel $k(x,x') = \exp(-\|x-x'\|_2^2/\gamma^2)$ with bandwidth parameter γ. Then MMD is the distance between the mean embedding of the distribution $\mathbb{P}$ and $\mathbb{Q}$ in RKHS. We denote the unbiased estimation of the MMD statistic as $\widehat{\mathrm{MMD}}_u^2(\mathbb{P},\mathbb{Q})$, which can be computed by

$$\frac{1}{m^2}\sum_{i,j=1;i\neq j}^{m} k(x_i,x_j) - \frac{2}{mn}\sum_{i,j=1}^{m} k(x_i,x'_j) + \frac{1}{n^2}\sum_{i,j=1;i\neq j}^{m} k(x'_i,x'_j).$$

Estimating the Null Distribution: It is complicated to build a test by directly estimating the distribution of MMD statistic in the null hypothesis [55]. We instead use the permutation test [85] to estimate the null distribution. This is a simpler and faster procedure if implemented carefully. According to previous work [31], under the null hypothesis, the instances in $\mathcal{X}$ and $\mathcal{X}'$ are exchangeable. Therefore, we can estimate the distribution of the MMD statistic under the null hypothesis by repeatedly calculating it with the instances randomly assigned to $\mathcal{X}$ and $\mathcal{X}'$ each time.

Asymptotic and Test Power of MMD: In this part, we discuss the relationship between sample size and test power. Generally, a larger sample size results in higher test power of MMD test with a fixed α. In other words, the TPR of our attribution verification will approach 100% with a fixed FPR if we have sufficient code snippets. This is theoretically guaranteed by the asymptotics of MMD as presented below:

Proposition 1 (Asymptotics of $\widehat{\mathrm{MMD}}_u^2$: under H_1). Under the alternative, $H_1 : \mathbb{P} \neq \mathbb{Q}$, a standard central limit theorem holds [78, Section 5.5.1]:

$$\sqrt{n}(\widehat{\mathrm{MMD}}_u^2 - \mathrm{MMD}^2) \xrightarrow{\mathrm{d}} \mathcal{N}(0,\sigma^2)$$
$$\sigma^2 := 4\left(\mathbb{E}[h_{12}h_{13}] - \mathbb{E}[h_{12}]^2\right),$$

where h_{12}, h_{13} refer to h_{ij} below:

$$h_{ij} := k(X_i,X_j) + k(X'_i,X'_j) - k(X_i,X'_j) - k(X'_i,X_j).$$

Proposition 1 proves that $\sqrt{n}(\widehat{\mathrm{MMD}}_u^2 - \mathrm{MMD}^2)$ converges in distribution to a normal distribution $\mathcal{N}(0,\sigma^2)$. Thus, the test power of MMD converges to a percentile of the standard normal

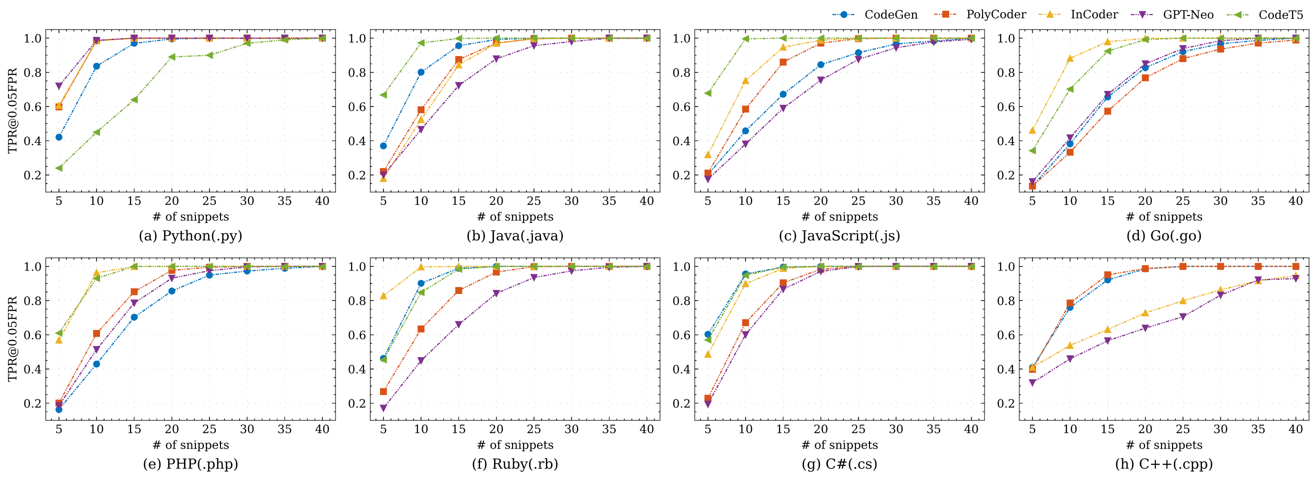


Fig. 9. Performance of attribution verification on different programming languages under different numbers of available testing code snippets.

distribution:

$$\Pr_{H_1}\left(n\widehat{\mathrm{MMD}}_u^2 > r\right) \xrightarrow{\mathbf{d}} \Phi\left(\frac{\sqrt{n}\,\mathrm{MMD}^2}{\sigma} - \frac{r}{\sqrt{n}\,\sigma}\right).$$

The approximate test power can be confirmed by using the rejection threshold r that can be determined by the permutation test mentioned above. According to [55], [85] and Proposition 1, it is also clear that the r will converge to a constant, and MMD, σ are also constants. Therefore, for a reasonably large n, the test power is dominated by the first term, i.e., $\sqrt{n}\,\mathrm{MMD}^2/\sigma$. Namely, for a reasonably large n, test power increases when n increases.

C. Experimental Setup

Like our previous experiments in Section V-A1, we utilize the pre-trained UniXcoder model to pre-process code data and obtain their embeddings as numerical features. We evaluate the test power (TPR) on 100 random subsets which are selected from the neural code datasets used in Section VI. We repeat this full process 10 times and report the average test power (TPR) of each test. When implementing the permutation test, we set the repeat times as 200 to estimate the distribution of MMD under the null hypothesis.

D. Results & Analysis

We now present the main results of the attribution verification. Fig. 9 illustrates the results of attribution verification using two-sample testing on different programming languages under different numbers of available testing instances. As expected, the number of available testing instances plays an important role in attribution verification, and the TPRs of attribution verification for all PLG models become larger with the increase in the number of testing code snippets. For example, we can observe that, in most cases, our attribution verification model achieves >80% TPR @ 5% FPR when the number of available testing instances increases to 20 or more, while the TPRs approach nearly 100% when there are more than 30 testing instances. This result is consistent with the asymptotics of the MMD statistic. In other words, given sufficient testing instances, the TPR of our two-sample testing based attribution verification can approach 100%.

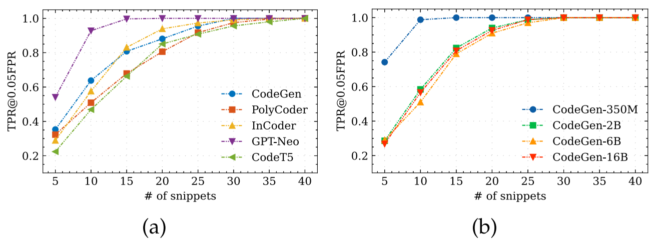


Fig. 10. (a) Performance of attribution verification when code snippets in $\mathcal{X}$ and $\mathcal{X}'$ are generated with prompts from APPS and MBPP, respectively. (b) Performance of attribution verification when code snippets are generated by similar PLG models.

Generalization to Different Prompt Datasets: To evaluate the generalization ability of our proposed attribution verification method, we consider the scenario where the code snippets in $\mathcal{X}$ and $\mathcal{X}'$ are generated from different prompt datasets. In the evaluation, we utilize the MBPP and APPS datasets to generate code snippets for X and $\mathcal{X}'$, respectively. We report the experimental results in Fig. 10(a). We can observe that our attribution verification achieves comparable performance when the number of testing code snippets is larger than 15, compared to the results in Fig. 9(a). For example, the TPR @ 5% FPR on PolyCoder drops 30% when the number of testing code snippets is only 5, but it goes to 94.1% when there are 15 testing code snippets, which is only a 5.9% drop in TPR. Additionally, similar to the results on the in-distribution prompt dataset, our attribution verification method achieves >80% TPR @ 5% FPR on all five models when the testing code snippets are from a different prompt dataset as the number of testing code snippets increases to 20, while the TPRs on five models are all >95% when the number of testing instances is 30 or more. These results demonstrate the generalization ability of our proposed attribution verification method.

Distinguishing Models in a Family: Similar to the evaluation in Section VI-D, we evaluate the performance of our attribution verification on distinguishing similar models that are different only in model size but designed by the same algorithm. We use four CodeGen models with different model sizes. Fig. 10(b) illustrates the verification results under different numbers of available testing instances. We can observe that our attribution verification achieves >90% TPR @ 5% FPR for all four models when the number of available testing instances increases to 20. Moreover, we can find that the TPR of CodeGen-mono-350 M is much higher than those of the other three models. For example, the attribution verification of code snippets generated by CodeGen-mono-350 M yields 74.2% TPR @ 5% FPR, even when there are only 5 testing code snippets. This means that the code snippets from CodeGen-mono-350 M can be easily distinguished from those from the other models. We speculate the reason is that the poor code generation performance of CodeGen-mono-350 M makes its output distribution lying relatively far away from those of other larger models.

- By relaxing the constraints on the number of neural code snippets, we propose MMD-based attribution verification to reliably verify whether this set of neural code snippets is generated by a given PLG model.
- We theoretically prove that our attribution verification can approach a 100% TPR with a fixed FPR when we have sufficient neural code snippets.
- Extensive evaluations on five PLG models and eight programming languages suggest that around 30 neural code snippets are sufficient for reliable verification.

VIII. Related Work

In this section, we review the related work in terms of four aspects: language models of code, training data audit, detection and attribution of neural text, as well as authorship attribution of human-written source code.

A. Language Models of Code

Inspired by the great success of large language models like GPT-2 [69] for natural languages (NL), language models of programming language (PL) have recently shown tremendous promise in code understanding and generation tasks. The left-to-right nature of decoder-only models makes them effective for program synthesis tasks. Therefore, this line of research has seen huge progress with many recently proposed models such as CodeGPT [58], CodeX [25], PolyCoder [92], GPT-Neo [16], AlphaCode [53] and CodeGen [65]. Different from decoder-only models, encoder-only models (e.g., CodeBERT [30], CuBERT [49], and DISCO [29]) commonly utilize a bidirectional training objective. This makes them suitable for producing an effective representation of the whole code snippet for downstream code-related understanding tasks. Encoder-only models and decoder-only models favor understanding and generation tasks, respectively. In contrast, encoder-decoder models can well support both code understanding and generation tasks. For example, PLBART [8] is based on BART model [52] while PyMT5 [26] and CodeT5 [90] explores the T5 framework [70] for programming language.

B. Training Data Audit

The aim of the training data audit is to determine if a data instance or set was used in training the model. A way is membership inference, which predicts whether or not a single data instance is used to train a target model. Most of the existing studies on membership inference focus on classification models [23], [44], [81], [94]. Only a few studies pay attention to language generation models [41], [82]. Song et al. [82] mount membership inference on LSTM-based text-generation models while Hisamoto et al. [41] apply it to transformer-based sequence-to-sequence models. However, they either merely focus on shallow and simple models [82] or yield poor performance on large transformer-based language models [41]. Moreover, they require to train multiple shadow models to mimic the behavior of the target model on member and non-member data instances. Recently, dataset watermarking has been proposed for data ownership verification in image [74] and code [83] domains. Sun et al. propose an approach to trace the usage of open-source code in PLG models via dead-code insertion [83]. However, the inserted dead code lacks stealthiness and could be easily identified by human inspection or filtered out by static code analysis tools.

C. Detection and Attribution of Neural Text

It has been given a lot of attention in developing ML-based approaches to distinguish between synthetic and human-written texts [34], [47], [97]. He et al. propose a comprehensive evaluation framework across different detection methods [37]. Furthermore, given a neural text, recent work has further attempted to attribute the neural text to its source LM [62], [68], [87]. In general, due to domain specificity, these existing methods that focus on natural languages are not directly applicable to programming languages, leaving the detection and attribution of neural code as an important but unexplored area. Watermarking methods for natural language is another technique to solve this attribution problem via marking and tracing text provenance. Recently, Abdelnabi and Fritz [5] propose an encoder-decoder network for text watermarking to replace the unobtrusive words in the original text with other inconspicuous words or symbols. However, such replacements still debilitate the logical and semantic coherence of the modified phrases [93]. Additionally, their dataset-specific model has poor transferability on text content with different text styles or data domains.

D. Authorship Attribution of Human-Written Source Code

There is a rich body of literature on authorship attribution of human-written source code that aims to identify the author of a given piece of code from a predefined set of authors [6], [18]. We discuss a few classic papers here. Caliskan et al. [22] built a code stylometry feature set to represent coding style and learned a random forest for authorship attribution. Abuhamad et al. [6] trained fully-connected layers with coding style features extracted by TF-IDF. Bogomolov et al. [18] utilized a pre-trained code representation model [10] to extract deep features and predict the author of the snippet. Our study fills the gap in exploring the properties of neural code generated by PLG models and providing comprehensive analysis to trace their accountability, which differs from the aforementioned studies in authorship attribution.

IX. Conclusion

This paper presents the first comprehensive study on the accountability of PLG models, investigating the problem from both model development and deployment perspectives. A holistic framework is proposed for auditing the training data usage of PLG models, detecting neural code generated by PLG models, and determining its attribution to a source model. To audit training usage, we introduce a membership inference method based on the likelihood ratio test. We provide several concrete recommendations to enhance the transparency and accountability of PLG models from both technical and regulatory perspectives. For neural code detection, we propose a learning-based method that can distinguish between human-written code and neural code. Extensive evaluations demonstrate that there exist common fingerprints shared among different PLG models. These findings can help practitioners detect neural code, and mitigate the severe threats recently encountered by the community. For neural code attribution, we empirically and theoretically demonstrate the impossibility of reliably solving this one-to-one attribution problem. These findings shed light on the challenges of dissecting neural code accountability. To this end, we propose two feasible alternative methods, attribution classification and attribution verification, by relaxing the constraints on either the number of PLG models or the number of neural code snippets. Extensive evaluations show the efficacy and robustness of the proposed feasible methods. The implementations of the proposed framework are also encapsulated into a new artifact, CodeForensic, to foster further research and inform the technical community at large of the responsible use of generative AI.

References

[1] aiXcoder, 2022. [Online]. Available: https://www.aixcoder.com/en/#/
[2] Doe 1 et al v. github, inc. et al., 2022. [Online]. Available: https://dockets.justia.com/docket/california/candce/4:2022cv06823/403220
[3] Github copilot, 2022. [Online]. Available: https://copilot.github.com/
[4] Copyright office launches new artificial intelligence initiative. U.S. Copyright Office, 2023. [Online]. Available: https://copyright.gov/newsnet/2023/1004.html
[5] S. Abdelnabi and M. Fritz, "Adversarial watermarking transformer: Towards tracing text provenance with data hiding," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 121–140.
[6] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 101–114.
[7] A. Aghajanyan et al., "CM3: A causal masked multimodal model of the internet," 2022, *arXiv:2201.07520*.
[8] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2021, pp. 2655–2668.
[9] H. Alkaissi and S. I. McFarlane, "Artificial hallucinations in ChatGPT: Implications in scientific writing," *Cureus*, vol. 15, 2023, Art. no. e35179.
[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, 2019, Art. no. 40.
[11] M. Artetxe et al., "Efficient large scale language modeling with mixtures of experts," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2022, pp. 5547–5569.
[12] B. Athiwaratkun et al., "Multi-lingual evaluation of code generation models," 2022, *arXiv:2210.14868*.
[13] J. Austin et al., "Program synthesis with large language models," 2021, *arXiv:2108.07732*.
[14] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2000, pp. 932–938.
[15] S. Biderman and E. Raff, "Fooling MOSS detection with pretrained language models," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2022, pp. 2933–2943.
[16] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow (v1.1.1). Zenodo.," 2021. doi: 10.5281/zenodo.5551208
[17] J. Bloch and P. Samuelson, "Some misconceptions about software in the copyright literature," in *Proc. Symp. Comput. Sci. Law*, 2022, pp. 131–141.
[18] E. Bogomolov, V. Kovalenko, Y. Rebryk, A. Bacchelli, and T. Bryksin, "Authorship attribution of source code: A language-agnostic approach and applicability in software engineering," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 932–944.
[19] R. Bommasani et al., "On the opportunities and risks of foundation models," 2021, *arXiv:2108.07258*.
[20] A. Borji, "A categorical archive of ChatGPT failures," 2023, *arXiv:2302.03494*.
[21] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, Art. no. 159.
[22] A. Caliskan-Islam et al., "De-anonymizing programmers via code stylometry," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 255–270.
[23] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer, "Membership inference attacks from first principles," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1897–1914.
[24] C. Chen, J. Fu, and L. Lyu, "A pathway towards responsible AI generated content," 2023, *arXiv:2303.01325*.
[25] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
[26] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: Multi-mode translation of natural language and python code with transformers," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2020, pp. 9052–9065.
[27] T. Davis, "DocSparse on twitter," Oct. 2022. [Online]. Available: https://twitter.com/DocSparse/status/1581461734665367554?cxt=HHwWhMDRibPEvfIrAAAA
[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.
[29] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Towards learning (dis)-similarity of source code from program contrasts," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2022, pp. 6300–6312.
[30] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Int. Conf. Findings Assoc. Comput. Linguistics*, 2020, pp. 1536–1547.
[31] V. A. Fernández, M. J. Gamero, and J. M. Garcia, "A test for the two-sample problem based on empirical characteristic functions," *Comput. Statist. Data Anal.*, vol. 52, pp. 3730–3748, 2008.
[32] D. Fried et al., "InCoder: A generative model for code infilling and synthesis," in *Proc. Int. Conf. Learn. Representations*, 2023, pp. 1–26.
[33] L. Gao et al., "The Pile: An 800GB dataset of diverse text for language modeling," 2020, *arXiv:2101.00027*.
[34] S. Gehrmann, H. Strobelt, and A. M. Rush, "GLTR: Statistical detection and visualization of generated text," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 111–116.
[35] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *J. Mach. Learn. Res.*, vol. 13, pp. 723–773, 2012.
[36] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2022, pp. 7212–7225.
[37] X. He, X. Shen, Z. Chen, M. Backes, and Y. Zhang, "MGTBench: Benchmarking machine-generated text detection," 2023, *arXiv:2303.14822*.
[38] M. Heikkilä, "Openai's hunger for data is coming back to bite it," Apr. 2023. [Online]. Available: https://www.technologyreview.com/2023/04/19/1071789/openais-hunger-for-data-is-coming-back-to-bite-it/

[39] P. Henderson, X. Li, D. Jurafsky, T. Hashimoto, M. A. Lemley, and P. Liang, “Foundation models and fair use,” 2023, *arXiv:2303.15715*.

[40] D. Hendrycks et al., “Measuring coding challenge competence with apps,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2021, pp. 1–11.

[41] S. Hisamoto, M. Post, and K. Duh, “Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system?,” *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 49–63, 2020.

[42] F. Hoffa, “GitHub on BigQuery: Analyze all the open source code,” Google, 2016. [Online]. Available: https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code

[43] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The curious case of neural text degeneration,” in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–16.

[44] H. Hu, Z. Salcic, L. Sun, G. Dobbie, P. S. Yu, and X. Zhang, “Membership inference attacks on machine learning: A survey,” *ACM Comput. Surv.*, vol. 54, 2022, Art. no. 235.

[45] HuggingFace, “Perplexity of fixed-length models,” 2022. [Online]. Available: https://huggingface.co/docs/transformers/perplexity

[46] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet challenge: Evaluating the state of semantic code search,” 2019, *arXiv: 1909.09436*.

[47] D. Ippolito, D. Duckworth, C. Callison-Burch, and D. Eck, “Automatic detection of generated text is easiest when humans are fooled,” in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 1808–1822.

[48] A. Jagannatha, B. P. S. Rawat, and H. Yu, “Membership inference attack susceptibility of clinical language models,” 2021, *arXiv:2104.08305*.

[49] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 5110–5121.

[50] N. Kokhlikyan et al., “Captum: A unified and generic model interpretability library for pytorch,” 2020, *arXiv: 2009.07896*.

[51] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi, “CodeRL: Mastering code generation through pretrained models and deep reinforcement learning,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2022, pp. 21314–21328.

[52] M. Lewis et al., “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 7871–7880.

[53] Y. Li et al., “Competition-level code generation with alphacode,” *Science*, vol. 378, pp. 1092–1097, 2022.

[54] Z. Li, G. Chen, C. Chen, Y. Zou, and S. Xu, “RoPGen: Towards robust code authorship attribution via automatic coding style transformation,” in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1906–1918.

[55] F. Liu, W. Xu, J. Lu, G. Zhang, A. Gretton, and D. J. Sutherland, “Learning deep kernels for non-parametric two-sample tests,” in *Proc. Int. Conf. Mach. Learn.*, 2020, Art. no. 586.

[56] J. R. Lloyd and Z. Ghahramani, “Statistical model criticism using kernel two sample tests,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 829–837.

[57] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–18.

[58] S. Lu et al., “CodeXGLUE: A machine learning benchmark dataset for code understanding and generation,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2021, pp. 1–16.

[59] C. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[60] F. Mireshghallah, K. Goyal, A. Uniyal, T. Berg-Kirkpatrick, and R. Shokri, “Quantifying privacy risks of masked language models using membership inference attacks,” in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2022, pp. 8332–8347.

[61] F. Mireshghallah, A. Uniyal, T. Wang, D. K. Evans, and T. Berg-Kirkpatrick, “An empirical analysis of memorization in fine-tuned autoregressive language models,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2022, pp. 1816–1826.

[62] S. Munir, B. Batool, Z. Shafiq, P. Srinivasan, and F. Zaffar, “Through the looking glass: Learning to attribute synthetic text generated by language models,” in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2021, pp. 1811–1822.

[63] S. K. Murakonda, R. Shokri, and G. Theodorakopoulos, “Quantifying the privacy risks of learning high-dimensional graphical models,” in *Proc. Int. Conf. Artif. Intell. Statist.*, 2021, pp. 2287–2295.

[64] J. Neyman and E. S. Pearson, “On the problem of the most efficient tests of statistical hypotheses,” *Philos. Trans. Roy. Soc. London*, vol. 231A, pp. 289–338, 1933.

[65] E. Nijkamp et al., “CodeGen: An open large language model for code with multi-turn program synthesis,” 2022, *arXiv:2203.13474*.

[66] OpenAI, “ChatGPT: Optimizing language models for dialogue?,” Nov. 2022. [Online]. Available: https://openai.com/blog/chatgpt/

[67] S. Overflow, “Temporary policy: ChatGPT is banned,” Dec. 2022. [Online]. Available: https://meta.stackoverflow.com/questions/421831/temporary-policy-chatgpt-is-banned

[68] X. Pan, M. Zhang, S. Ji, and M. Yang, “Privacy risks of general-purpose language models,” in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1314–1331.

[69] A. Radford et al., “Language models are unsupervised multitask learners,” OpenAI blog, vol. 1, no. 8, pp. 9, 2019.

[70] C. Raffel et al., “Exploring the limits of transfer learning with a unified text-to-text transformer,” *J. Mach. Learn. Res.*, vol. 21, 2020, Art. no. 140.

[71] V. Raychev, P. Bielik, and M. Vechev, “Probabilistic model for code with decision trees,” *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[72] J. Rodriguez, T. Hay, D. Gros, Z. Shamsi, and R. Srinivasan, “Cross-domain detection of GPT-2-generated technical text,” in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2022, pp. 1213–1233.

[73] A. Ronacher, “Mitsuhiko on twitter,” Jul. 2021. [Online]. Available: https://twitter.com/mitsuhiko/status/1410886329924194309

[74] A. Sablayrolles, M. Douze, C. Schmid, and H. Jégou, “Radioactive data: Tracing through training,” in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 8326–8335.

[75] P. Samuelson and C. D. Asay, “Saving software’s fair use future,” *Harv. JL & Tech.*, vol. 31, pp. 535, 2017.

[76] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 76–85.

[77] B. Schölkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, “Support vector method for novelty detection,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 1999, pp. 582–588.

[78] R. J. Serfling, *Approximation Theorems of Mathematical Statistics*. Hoboken, NJ, USA: Wiley, 1980.

[79] J. Serrà, D. Álvarez, V. Gómez, O. Slizovskaia, J. F. Núñez, and J. Luque, “Input complexity and out-of-distribution detection with likelihood-based generative models,” in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–15.

[80] X. Shen, Z. Chen, M. Backes, and Y. Zhang, “In ChatGPT we trust? Measuring and characterizing the reliability of ChatGPT,” 2023, *arXiv:2304.08979*.

[81] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership inference attacks against machine learning models,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 3–18.

[82] C. Song and V. Shmatikov, “Auditing data provenance in text-generation models,” in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 196–206.

[83] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, “CoProtector: Protect open-source code against unauthorized training usage with data poisoning,” in *Proc. ACM Web Conf.*, 2022, pp. 652–660.

[84] M. Sundararajan, A. Taly, and Q. Yan, “Axiomatic attribution for deep networks,” in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 3319–3328.

[85] D. J. Sutherland et al., “Generative models and model criticism via optimized maximum mean discrepancy,” in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1–11.

[86] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing With Transformers*. Sebastopol, CA, USA: O’Reilly Media, Inc., 2022.

[87] A. Uchendu, T. Le, K. Shu, and D. Lee, “Authorship attribution for neural text generation,” in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2020, pp. 8384–8395.

[88] L. Van der Maaten and G. Hinton, “Visualizing data using t-SNE,” *J. Mach. Learn. Res.*, vol. 9, pp. 2579–2605, 2008.

[89] B. Wang and A. Komatsuzaki, “GPT-J-6B: A 6 billion parameter autoregressive language model,” 2021. [Online]. Available: https://github.com/kingoflolz/mesh-transformer-jax#gpt-j-6b

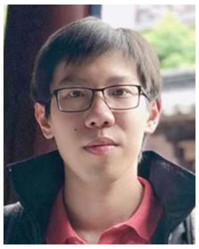
[90] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[91] Z. Wang, B. Dai, D. Wipf, and J. Zhu, “Further analysis of outlier detection with deep generative models,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 11–20.

[92] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, 2022, pp. 1–10.
[93] X. Yang et al., "Tracing text provenance via context-aware lexical substitution," in *Proc. AAAI Conf. Artif. Intell.*, 2022, pp. 11613–11621.
[94] J. Ye, A. Maddi, S. K. Murakonda, V. Bindschaedler, and R. Shokri, "Enhanced membership inference attacks against machine learning models," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 3093–3106.
[95] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha, "Privacy risk in machine learning: Analyzing the connection to overfitting," in *Proc. IEEE 31st Comput. Secur. Found. Symp.*, 2018, pp. 268–282.
[96] D. Zan et al., "CERT: Continual pre-training on sketches for library-oriented code generation," in *Proc. Int. Joint Conf. Artif. Intell.*, 2022, pp. 2369–2375.
[97] R. Zellers et al., "Defending against neural fake news," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 812.
[98] C. Zhang et al., "One small step for generative AI, one giant leap for AGI: A complete survey on ChatGPT in AIGC era," 2023, *arXiv:2304.06488*.
[99] L. Zhang, M. Goldstein, and R. Ranganath, "Understanding failures in out-of-distribution detection with deep generative models," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 12427–12436.

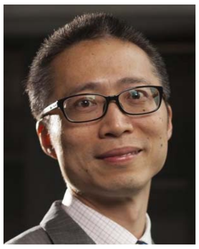
Wanlun Ma received the bachelor's and master's degrees in communication engineering from the University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 2017 and 2020, respectively. He is currently working toward the PhD degree with the School of Software and Electric Engineering, Swinburne University of Technology. His research interests focus on adversarial machine learning, deep neural networks, and online social network security.

Yiliao Song (Member, IEEE) received the MS degree in probability and statistics in mathematics from the School of Mathematics and Statistics, Lanzhou University, China, in 2015, and the PhD degree in computer science from the University of Technology Sydney, Australia. Her research interests include concept drift, data stream mining, and real-time prediction. She is a research fellow with RMIT Enterprise AI and Data Analytics Hub, RMIT University, Australia.



Minhui Xue (Member, IEEE) is a senior research scientist (lead of AI Security sub-team) with CSIRO's Data61, Australia. His current research interests include machine learning security and privacy, system and software security, and Internet measurement. He is the recipient of the ACM CCS Best Paper Award Runner-Up, ACM SIGSOFT distinguished paper award, Best Student Paper Award, and the IEEE best paper award, and his work has been featured in the mainstream press, including The New York Times, Science Daily, PR Newswire, Yahoo, The Australian Financial Review, and The Courier. He currently serves on the Program Committees of IEEE Symposium on Security and Privacy (Oakland) 2023, ACM CCS 2023, USENIX Security 2023, NDSS 2023, ACM/IEEE ICSE 2023, and ACM/IEEE FSE 2023. He is a member of the ACM.

Sheng Wen received the PhD degree from Deakin University, Australia, in 2014. He is currently working full-time as an associate professor with the Swinburne University of Technology. He is also the director of Blockchain Innovation Lab and the deputy director of Swinburne Cybersecurity Lab, Swinburne University. In addition, he has published more than 100 high-quality papers, including top conference papers such as papers in IEEE S $\&amp;$ P, ACM CCS, USENIX Sec, NDSS, ICSE, FSE, and WWW, as well as many papers in IEEE/ACM transactions series journals. Some of these papers are with high impact factors or become highly cited papers. His research interests include social network analysis and system security.



Yang Xiang (Fellow, IEEE) received the PhD degree in computer science from Deakin University, Burwood, VIC, Australia, in 2007. He is currently a full professor and the dean of Digital Research with Swinburne University of Technology, Hawthorn, VIC, Australia. He is also leading the Blockchain initiatives at Swinburne. In the past 20 years, he has published more than 300 research papers in many international journals and conferences. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is the editor-in-chief of the *Springer Briefs on Cyber Security Systems and Networks*. He serves as an associate editor of the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal*, *ACM Computing Surveys*, *IEEE Transactions on Computers* and *IEEE Transactions on Parallel and Distributed Systems*.