Hiding in Plain Sight: On the Robustness of AI-generated Code Detection

Saman Pordanesh, Sufiyan Bukhari[0009−0008−5103−9067], Benjamin Tan[0000−0002−7642−3638], and Lorenzo De Carli[0000−0003−0432−3686]

University of Calgary, Calgary AB T2N 1N4, Canada
{saman.pordanesh, sufiyanahmed.bukhari, benjamin.tan1, lorenzo.decarli@ucalgary.ca✉}

Abstract. AI code assistants, such as GitHub Copilot, are an increasingly popular coding aid, but they also present risks. Large language models (LLMs) upon which those assistants are built may generate insecure/incorrect code, either by accident or as a result of code poisoning attacks. In general, LLMs obfuscate the lineage of source code used for training. This is a problem, for example, in the context of supply chain security, where tracking provenance is of the utmost importance. While a number of recent approaches can flag AI-generated code based on a combination of lexical and syntactic features, such works have not been evaluated in realistic settings. First, we identify and operationalize a number of recently proposed AI code identification tools, measuring their baseline performance on datasets generated by state-of-the-art models. Then, we verify the robustness of such approaches to variations in training sets and prompting strategies. Results show that existing AI code detectors tend to be fragile and have limited accuracy in real-world scenarios.

1 Introduction

AI code assistants are quickly becoming ubiquitous within the software development cycle. Such tools can quickly generate source code on demand, either by completing developer-written code or in response to specific user requests. In doing so, they can greatly reduce the effort to write boilerplate code, tests, and similar components. This, in turn, reduces development time and frees programmers to concentrate on higher-level tasks, such as debugging and extending functionality. As such, these tools enjoy ever-increasing popularity [31].

While AI code assistants have the potential to be immensely useful, they also give rise to security concerns. Recent research points to the fact that code assistants may, in certain situations, generate code that is less secure than that written by humans [26, 27]. Further, Large Language Models (LLMs), on which assistants are based, are trained on large datasets of unvetted Open-Source code. As such, they may learn to generate copyrighted [8] or incorrect code, or even be the target of code poisoning attacks [30, 39, 41]. This is potentially problematic, both in the context of direct use of the tools and when importing software dependencies that may be AI-generated. Overall, LLMs obfuscate provenance [5],

as code generation is based on a training dataset, but the relation between training and output is not clearly maintained. Thus, being able to track *code provenance* back to a code assistant is of the utmost importance for security, correctness, and legal reasons. Indeed, it is not uncommon for software companies to limit or qualify the use of such tools by their employees [13, 29]. For the reasons above, until the threat model surrounding AI-generated code is better understood, there is a need for tools that can highlight the presence of AI-generated code in the wild so that it can be appropriately reviewed if necessary.

Several recent works propose the design of classifiers that can identify the human or AI provenance of source code with high accuracy, at least under certain assumptions [5,17,22,32,35,40]. We believe such algorithms can fill an important gap and be useful for AI code detection and general code measurement studies. Unfortunately, the robustness of such tools has seen limited to no investigation. There are multiple threats to their accuracy. One is overfitting the training set: as multiple LLMs capable of generating code exist, there may be intrinsic differences in the code they generate. This may make a detector trained on one model underperform on code generated by another. Another issue is that differences in the characteristics of human and AI-generated code may depend on specific programming tasks. Indeed, past work observed that "[detector] performance considerably improves when the common patterns – those that may occur in data curated from the same domains – have been learned during the training" [22].

In this paper, we examine the robustness of recently proposed AI code detectors. We consider multiple classifiers [5, 22, 32, 41]. First, we measure baseline classifier performance and examine the impact of classifier design parameters. Then, we evaluate the effect of the classifier training dataset (where applicable) and prompt variations on classifier effectiveness. Finally, we consider whether diversifying training sets can help improve classifier performance.

For evaluation accuracy, it is important to use cleanly labeled, diverse datasets. We use a set of 6K software samples from two source code datasets commonly used in this domain [15, 16], each including programming task assignments and human-generated solutions. To maximize external validity, we generate corresponding AI solutions using three prominent models: OpenAI's GPT [24], Google's Gemini [28], and Anthropic's Claude [2]. Furthermore, there exist numerous AI code detectors based on different approaches. Most only have proof-of-concept implementations, and for some, a full implementation has not been released. We perform a literature review, selecting four representative detectors (see Section 4) [5, 22, 32, 41]. We engineer all these approaches to perform classification on our dataset, implementing missing components when necessary[1].

Overall, zero-shot detectors tend to exhibit a significant drop in accuracy compared to the originally published results, even after parameter tuning. Classifiers based on train+evaluate ML pipelines tend to fare better, but this advantage dissipates when training and evaluation sets have different characteristics. These results suggest that the problem of AI-generated code detection is not trivially solvable, and more work is needed to produce effective detectors.

[1] Data and code package: `https://osf.io/jahxs/?view_only=dff479fdad8c4b9cb2060e6a3c2c5e5a`

2 Background

2.1 AI Code Assistant

We use the term "AI code assistant" to refer to any AI-based tool that can generate code on demand. We include in our definition both tools that are directly integrated into software development IDEs, such as GitHub Copilot [11], and tools that offer some form of chat UI where users can post questions and retrieve code from the answers, such as OpenAI's ChatGPT [24]. These code assistants are generally based on LLMs. In extreme synthesis, those are transformer models with billions of parameters trained on petabytes of human-generated text. They have the ability to continue a prompt with a stream of words (tokens) likely to follow, thus generating, in many circumstances, "human-like" answers to queries. In the coding domain, those models are used to complete source code or to generate source code from scratch based on natural language descriptions.

The training dataset of prominent "foundation" LLMs includes many forms of text, including, in the case of models used for coding, a substantial amount of source code. Due to its size, text in the training data is often used as-is without vetting for correctness or risks. In many cases, the specifics of those training datasets are not made public. This has raised a number of concerns, including the possibility of bias and/or generation of technically incorrect information [38], and vulnerability of LLMs to poisoning attacks [30].

2.2 Supply Chain Security

Software supply chain security focuses on identifying risks and vulnerabilities in components used within software artifacts. Much modern software is built compositionally, importing open-source software (OSS) components within a project. These components provide ready-made, freely available implementations of functionality that may be tedious or complex to implement correctly (e.g., JSON parsers, messaging middleware, web UI frameworks, etc.). In the last 15 years, this approach has revolutionized software development, enabling companies to quickly and economically build infrastructure that previously would have needed to be entirely developed in-house.

However, this approach also comes with security drawbacks, as including a large amount of external modules can significantly extend the attack surface of an software artifact [36]. AI coding assistants further complicate the issue: as discussed in Section 1, they may obfuscate the provenance of code within imported modules, and may even introduce additional vulnerabilities. Thus, we consider AI code detection relevant to software supply chain security.

2.3 Threat Model

We consider a scenario where an AI code assistant generates vulnerable or malicious code that gets embedded in a software artifact. This may be accidental, e.g., due to defective examples in the training set, or purposefully caused by an

Prompt (abridged): "Utility function to calculate md5 hashes while being light on memory usage. By reading the fileobject piece by piece, we are able to process content that is larger than available memory"

(a) Prompt used to generate the function

```python
def calculate_md5(fileobject, size=2**16):
  fileobject.seek(0)
  md5 = hashlib.md5()
  for data in iter(lambda: fileobject.read(size), b''):
    if not data: break
    if isinstance(data, six.text_type):
      data = data.encode('utf-8') # md5 needs a byte string
    md5.update(data)
  fileobject.seek(0) # rewind read head
  return md5.hexdigest()
```

(b) Human-generated example

```python
def calculate_md5(fileobject):
  md5 = hashlib.md5()
  while chunk := fileobject.read(8192):
    md5.update(chunk)
  return md5.hexdigest()
```

(c) AI-generated example

Fig. 1: Example of prompt and human- and AI-generated code (CSN dataset)

attacker conducting dataset poisoning [30,39]. To contain such attacks, an organization may opt to identify and track the provenance of AI-generated code to a specific model/training set. This may enable countermeasures such as only allowing the use of models in non-security-sensitive contexts or limiting code generation to specific, vetted models/training sets. To do so, we consider the problem of identifying AI-generated code and distinguishing it from human-generated code. While classifiers have been proposed for this problem [5,22,32,41], as part of a more general trend towards distinguishing AI- and human-generated content [12,21], these tools have only been evaluated in "clean lab" settings, typically on limited datasets and in the absence of confounding factors that exist in the real world (e.g., training and evaluation sets with different characteristics) and may degrade their performance. In this work, we aim to evaluate and compare the performance of different classifiers in the presence of such factors.

3 Dataset

3.1 Prompt corpora

To assess the effectiveness of classifiers, we required a diverse dataset containing code samples generated by both humans and AI. To control confounding factors, we sought to have AI generate code for tasks for which corresponding human implementations exist. To ensure and maintain the integrity of the dataset generation process, we established specific criteria for our dataset selection.

Comprehensive Problem Descriptions: Each human-written code sample must be accompanied by a detailed problem statement. This context is crucial for a fair comparison between human and AI-generated code. Without it, AI may produce incorrect or irrelevant code, potentially leading to biased analysis.

Sufficient Number of Code Samples: The dataset should contain a substantial number of code samples to enable large-scale analysis. A larger dataset leads to more reliable and robust conclusions.

As the target language, we choose Python due to (i) the large amount of Python code available for model training, which makes widely available AI code assistants particularly suited to Python programming (Python routinely tops

Prompt (abridged): "Write a program which prints multiplication tables in the following format: 1x1=1 1x2=2 [...] [template for various languages follows]"

(a) Prompt used to generate the function

```
def main():
    for a in range(1,10):
        for b in range(1,10):
            print("{}x{}={}".format(a,b,a*b))
    return None
```

(b) Human-generated example

```
def print_multiplication_table():
    for i in range(1, 10):
        for j in range(1, 10):
            print(f"{i}x{j}={i*j}")

print_multiplication_table()
```

(c) AI-generated example

Fig. 2: Example of prompt and human- and AI-generated code (IBM dataset)

lists of most popular programming languages [34]); and (ii) the widespread use of Python in literature on AI-generated code detection [32, 41].

CodeSearchNet The CodeSearchNet Corpus [15] is a collection of 6M+ code functions from popular open-source GitHub repositories. They are written in a variety of programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby. In the rest of the paper, we refer to this as the **CSN dataset**.

The dataset comprises approximately 2 million pairs of code functions and their corresponding documentation. Additionally, it includes around 4 million functions without associated documentation to facilitate model training and evaluation for training, validation, and testing.

The primary goal of this dataset is to support the CodeSearchNet challenge. This challenge focuses on the development of advanced code search techniques that can accurately retrieve code snippets based on natural language queries.

Prompt set review. Manual review of this dataset reveals potential issues. Documentation is oftentimes truncated, resulting in loss of contextual information. In some cases, the documentation fails to reflect requirements that are clear in the human implementation, and/or the documentation is outdated and no longer accurately reflects the original code. Figure 1 presents an example, together with a human solution and a sample AI solution generated with OpenAI GPT 4o. It is worth noting that while this dataset is popular in the AI code detection literature [32, 41], these issues are virtually ignored. Ultimately, we decided to use this dataset as, despite its limitations, it has the advantage of comprising realistic software development tasks while noting that the information asymmetry between human developers and AI may introduce systematic differences.

IBM Project CodeNet Project CodeNet [16], by IBM, is a massive dataset of 13.9 million code samples, each designed to solve one of 4,000 coding challenges. Sourced from AIZU Online Judge [2] and AtCoder [3], these samples cover over 50 programming languages, primarily C++, C, Python, and Java. Each sample includes detailed metadata like size, memory usage, execution time, and outcome

[2] `https://onlinejudge.u-aizu.ac.jp/home`
[3] `https://atcoder.jp`

```
Provide python code as a function for the below problem statement and
produce no other text. Do not include the function inside a docstring.

Problem Statement:

PROBLEM DESCRIPTION GOES HERE.
```

Fig. 3: Template used for AI code generation

(accepted or rejected with reason). Human code samples are pre-vetted by human judges. In the following discussion, we refer to this as the **IBM dataset**.

Over 90% of the problems in Project CodeNet are accompanied by detailed descriptions. Additionally, over half of the 4K problems in the dataset have at least one accepted code solution, providing a benchmark for evaluation.

Problem set review. Different from CodeSearchNet, samples in this dataset do not suffer from information asymmetry: human samples were developed using exactly the same information available for AI code generation. These descriptions outline the specific problem and expected input and output formats, often including sample input/output pairs. This rich contextual information provides a comprehensive understanding of the problem to be solved. Figure 2 presents an example, together with a human solution and a sample AI solution generated with OpenAI GPT 4o. However, this dataset suffers from a different limitation, as prompts veer towards artificial programming exercises rather than real-world tasks. We decided to still retain this dataset, as it has the advantage of providing clear, unambiguous tasks together with rigorously vetted human solutions.

3.2 Dataset generation and discussion

Human-generated samples We identified potential prompts for the LLMs (coding task *problem descriptions* for the AI) in both CSN and IBM sets, requiring (i) the prompt to be in English (to eliminate language as a confounding factor); (ii) for the IBM prompt set, an accepted solution to be available; and (iii) the solution to be non-empty and valid Python (as some classifiers generate features based on the extracted Abstract Syntax Tree, which requires syntactic validity). Finally, we randomly sampled a set of approximately 3000 solutions from the acceptable samples for each dataset. As discussed in Section 3.3, we only retained human samples for which a corresponding machine-generated solution could be obtained from all models. This resulted in **3042** code/prompt samples for CSN and **2984** code/prompt samples for IBM.

3.3 AI-generated samples

To construct code-generation prompts, we used the same problem descriptions present in the set of human samples (described above) for both the CSN and IBM datasets. We selected three state-of-the-art enterprise LLMs as our generative

	GPT		Gemini		Claude	
	CSN	IBM	CSN	IBM	CSN	IBM
Length	0.06	0.76	0.66	0.96	0.70	0.97
Cyclomatic Comp.	0.31	-0.20	0.48	-0.28	0.23	-0.26

Table 1: Effect size for Wilcoxon pairwise comparison between length and complexity of human and AI samples (strong correlation highlighted)

models for addressing code-related problems: GPT-4o [24], Gemini 1.5 Flash [28], and Claude 3.5 Sonnet [2], based on the most recent versions available as of mid-2024. Despite their strong performance, open-source models such as LLaMA 3.0 [20] were excluded. They are hosted on limited cloud infrastructures and are, therefore, less commonly used for day-to-day tasks like programming. All code generation processes using these models were conducted via their APIs.

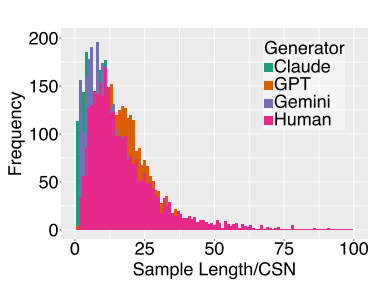
Generation Process We created a two-part initial prompt template to guide the generative models. The first part provided specific instructions about the desired response format and content, while the second part incorporated a problem description from one of the prompt sets. The prompt was created by following OpenAI's suggested best practices [25] and iteratively refined to minimize invalid/incorrect code generation in initial informal experimentation. It also includes instructions to minimize the generation of extraneous non-code text. The prompt is given in Figure 3. We further discuss the potential impacts of prompt structure on detection results in Section 5.3.

After querying the generative models, the generated samples were checked for non-emptiness and correctness. When checks failed, we re-tried code generation up to three times, after which we dropped the prompt from the evaluation set. Additionally, we observed that AI-generated samples occasionally only consist of documentation (i.e., docstrings); we removed all such documentation before checking for empty samples.

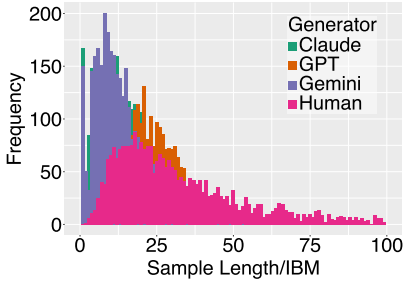
During generation, all LLMs failed to generate code for several problem description prompts; the set of failed prompts overlaps only partially. To ensure a fair evaluation across models, we only retain prompts/generated code for prompts from which all models could generate a valid code sample.

Quantitative datasets review Observing information asymmetry in CodeSearchNet raises the question of whether systematic differences between human- and AI-generated code exist. To address it, we compute two basic measures between paired code samples from our dataset: lengths in lines of code (LOCs), and cyclomatic complexity [19]. The latter is defined as the number of independent paths through a source code artifact and is frequently used to quantitatively represent the level of complexity of a source code artifact.

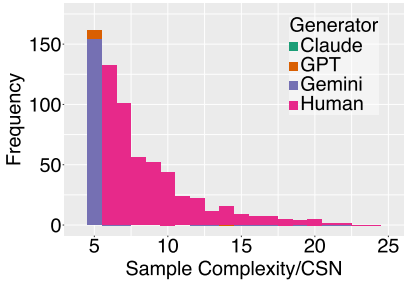
Distribution of sample lengths from samples generated by human programmers, and the GPT, Gemini and Claude models on CodeSearchNet and IBM are depicted in Figure 4(a) and (b), while complexity distributions are depicted in Figure 4(c) and (d). The plots present a nuanced picture, showing



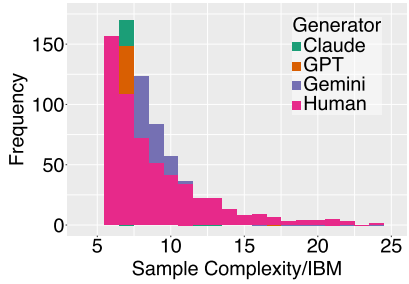
(a) Distribution of sample lengths (CSN)



(b) Distribution of sample lengths (IBM)



(c) Distribution of complexity lengths (CSN)



(d) Distribution of sample complexity (IBM)

Fig. 4: Characterization of basic dataset metrics

small but noticeable distributional differences. Motivated by this, we conducted Wilcoxon signed-rank tests pairwise between human samples and each set of model-generated samples. All metrics are found to present statistically significant differences at $p <= 0.01$. Effect sizes are measured using Rank Biserial Correlation (RBC) (r). Following practices from the literature, we interpret r values above 0.6 as representing high correlation. Full results are given in Table 1. Overall, these results suggest macroscopic statistical differences between samples generated by humans and most sets of AI-generated samples. While this does not imply that individual samples may be correctly labeled, it suggests that there are indeed systemic effects at play that may make classification possible.

4 Evaluated Classifiers

To identify suitable classifiers from the analysis, we reviewed the literature on the detection of AI-generated code. We did so by reviewing publications in high-profile security venues, such as IEEE S&P, ACM CCS, and similar, searching the arXiv online archive, and using search engines (e.g., Google Scholar) to identify any missing publications. We only retained publications with partial or full code released and sufficient details to achieve a working implementation on our

dataset with reasonable effort. We further filtered approaches for which we could not get accuracy and or F1 score consistently above 60%. For example, we were unable to replicate Yang et al. [40] 70-80% AUC on Python samples in our experiments. Finally, we filtered approaches that used similar techniques, retaining one example per category. For example, we did not evaluate Whodunit [17], as it is conceptually very similar to the one developed in our previous work [5], discussed below. We discussed the selected classifiers in the following.

4.1 Bukhari et al.

In our previous work [5] we evaluated a classifier based on syntactic and semantic features mutuated from the code stylometry feature set by Caliskan et al. [6]. Such features are computed from source code and an intermediate AST representation, and fed to a trained ML classifier. While the initial code release[4] uses a subset of the stylometric features by Caliskan et al., we were able to extend it with limited effort to the full feature set, thus more closely representing the potential of stylometry-based approaches. This classifier's performance are heavily dependent on which algorithm is used for classification (SVM, XGBoost, random forest etc.). Based on preliminary experiments, XGBoost returns the best result, and we use it for our experiments. For simplicity, in the rest of this paper we refer to this classifier as "Bukhari et al.".

4.2 GPTSniffer

GPTSniffer [22] is a CodeBERT-based classifier designed to detect source code generated by ChatGPT. By fine-tuning a pre-trained language model, it aims at identifiying patterns and anomalies specific to machine-generated code. This approach has a fully functional code release[5], only requiring to format our dataset appropriately. We further make minor alterations to the code to perform cross-validation, compute additional metrics, and add small quality-of-life improvements (e.g., saving the model after training for reuse).

4.3 Ye et al.

Ye et al. [41] introduce a zero-shot synthetic code detection technique through code rewriting. Specifically, this model measures how code properties change when a sample is partially rewritten by an AI code generation tool. This model has the least complete implementation, requiring a substantial effort to extend its code release[6] to a functional tool. We reimplemented missing components based on the description given in the paper. Further, this detector can be implemented in different ways, as it is heavily dependent on which model is internally used for rewriting, and how many times the code is regenerated. To provide

[4] `https://osf.io/46nva/?view_only=9110c4a94f0a4b4591f14fdd976deeca`
[5] `https://github.com/MDEGroup/GPTSniffer`
[6] `https://anonymous.4open.science/r/code-detection-6B35/README.md`

	GPT 4o					Gemini					Claude				
Approach	**P**	**R**	**A**	**F1**	**AUC**	**P**	**R**	**A**	**F1**	**AUC**	**P**	**R**	**A**	**F1**	**AUC**
Bukhari et al.	0.86	0.89	0.89	0.88	0.97	0.83	**0.83**	0.83	0.83	0.92	0.94	0.93	0.94	0.94	**0.99**
GPTSniffer	**0.95**	**0.98**	**0.97**	**0.97**	**0.99**	**0.84**	0.94	**0.89**	**0.89**	**0.95**	**0.93**	**0.98**	**0.95**	**0.95**	**0.99**
Ye et al.*	**0.58**	**0.58**	**0.58**	**0.58**	**0.6**	**0.5**	**1.00**	**0.5**	**0.67**	**0.44**	0.65	**0.62**	**0.65**	**0.63**	**0.69**
DetectCodeGPT*	0.61	0.85	0.85	0.71	0.72	0.51	0.97	0.74	**0.67**	0.63	**0.60**	0.85	0.79	0.7	0.72

Table 2: Performance of evaluated classifiers on CSN dataset using GPT-4o, Gemini Flash 1.5, and Claude Sonnet 3.5 for code generation. * denote zero-shot models which were evaluated on whole dataset; for other models, 4-fold cross-validation was used (**P:** Precision; **R:** Recall; **A:** Accuracy). Highest values in green, lowest in purple.

a fair evaluation, we experimented extensively with different models (including OpenAI GPT 4o Mini and Gemini 1.5 Flash), and 4/8/16 rewrites, and retained the combination offering the best results.

4.4 DetectCodeGPT

DetectCodeGPT [32] applies zero-shot machine-learning to distinguish between machine-generated and human-written code. This approach works by introducing perturbations in code samples, and measuring how these alter code *naturalness*. Like GPTSniffer, this approach has a fairly complete implementation[7], only requiring reformatting our dataset and computing additional metrics.

5 Experimental Evaluation

5.1 Research Questions

In this section, we seek to answer the following research questions:

- *RQ1: What are the baseline performance of evaluated classifiers on distinguishing AI- and human-generated code?* Section 5.2 demonstrates that classifiers exhibit varying degrees of accuracy.
- *RQ2: How robust are these classifiers to variations in training set and code generation prompts?* Experiments in Section 5.3 show that, in many cases, model accuracy decreases drastically when training samples come from a different problem domain than evaluation samples.
- *RQ3: Can classifier performance be improved by diversifying training set?* Experiments in Section 5.4 suggest that diversifying training set has limited impact on performance.

5.2 Baseline classifier performance

In this section, we evaluate baseline performance of the four classifiers under examination. For "baseline performance", we intend executing the classifiers in

[7] `https://github.com/YerbaPage/DetectCodeGPT`

	GPT 4o					Gemini					Claude				
Approach	**P**	**R**	**A**	**F1**	**AUC**	**P**	**R**	**A**	**F1**	**AUC**	**P**	**R**	**A**	**F1**	**AUC**
Bukhari et al.	0.94	0.95	0.97	0.95	1.00	0.94	0.95	0.97	0.95	0.99	0.95	0.96	0.98	0.96	**1.00**
GPTSniffer	**0.99**	**1.00**	**0.99**	**0.99**	**1.00**	**0.99**	**1.00**	**0.99**	**0.99**	**1.00**	**0.99**	**1.00**	**0.99**	**0.99**	**1.00**
Ye et al.*	0.63	**0.76**	0.65	0.69	0.71	**0.58**	**0.12**	**0.52**	**0.20**	**0.46**	0.67	**0.61**	0.66	**0.64**	0.72
DetectCodeGPT*	**0.51**	0.99	**0.51**	**0.67**	**0.51**	0.63	0.80	0.73	0.70	0.73	**0.53**	0.93	**0.60**	0.67	**0.59**

Table 3: Performance of evaluated classifiers on IBM dataset using GPT-4o, Gemini Flash 1.5, and Claude Sonnet 3.5 for code generation. * denote zero-shot models which were evaluated on whole dataset; for other models, 4-fold cross-validation was used (**P:** Precision; **R:** Recall; **A:** Accuracy). Highest values in green, lowest in purple.

the absence of confounding factors. These are primarily a concern in the case of training based classifiers (GPTSniffer and Bukhari et al.), as the accuracy of those may suffer when trained on samples whose characteristics differ from the evaluation set. Overall, this analysis also serves to establish baseline performance expectations on our dataset.

Table 2 and 3 presents baseline results for the five classifiers under examination. We do not report variations across cross-validation folds as for all metrics the range is at most 0.03, and typically well below. Training-based models (Bukhari et al. and GPTSniffer) generally perform reasonably well, with F1 scores substantially above 0.8 and in line with published results. However, there is the question of whether such results hold up when training and evaluation set have different charateristics, as samples derived from different sources may be distributionally different. We evaluate this question in the next subsection.

Interestingly, zero-shot models performance measured on our datasets were significantly lower than published results. Ye et al. [41] report AUCs above 80% for many experimental scenarios on APPS [14] and MBPP [3] benchmarks. We note that this approach performance appear sensitive to specific implementation choices, such as the number of rewrites and the model used for rewriting. We further investigate the impact of these choices below.

Similarly, DetectCodeGPT [32], which is a zero-shot classifier, shows varied base-line results through different models and dataset in comparison with the original paper. While the original publication was also based on the CSN dataset, it used relatively small models (1-7B parameters) for AI code generation, which may explain the discrepancies with our results.

Bukhari et al. Design Space Analysis Bukhari et al.'s approach works by (i) using AST-based analysis to transform each sample program in a feature vector; and (ii) training and using a Machine Learning classifier to label sample vectors as either human- or machine-generated. As such, it is sensitive to the particular algorithm used to train the classifier. We evaluate both XGBoost and Random Forest classifiers as those resulted in the best performance in their original paper [5]. F1 scores for both classifiers for all combinations of problem set/generator model (using cross-validation) are depicted in Figure 5. XGBoost results in marginally better performance, and we use it for all other experiments.

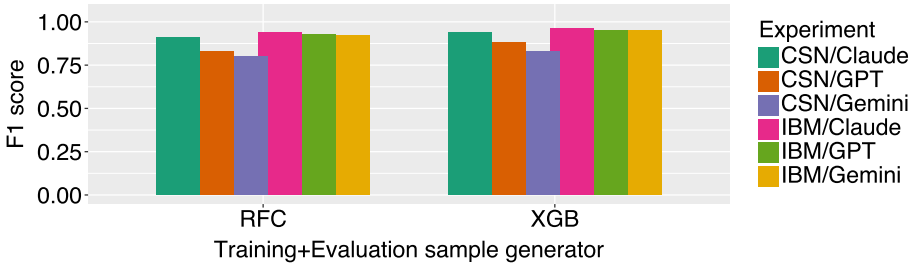


Fig. 5: F1 score for Bukhari et al. depending on classification algorithm used.

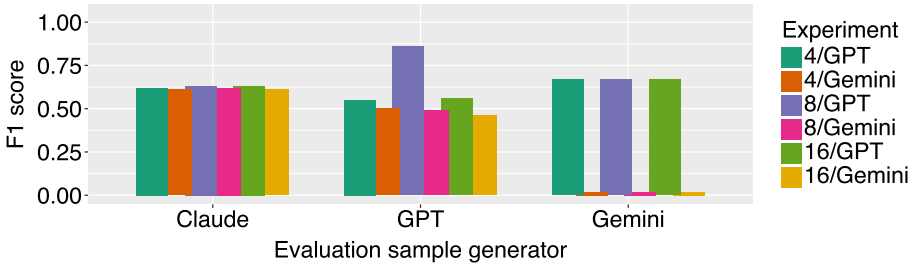
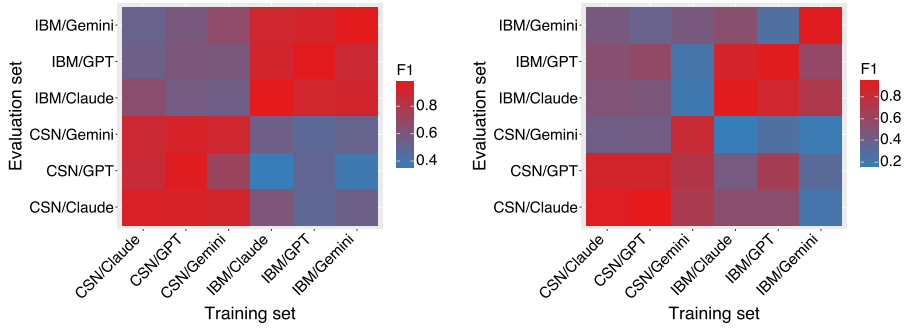


Fig. 6: F1 score for Ye et al. depending on number of rewrites/model used to generate rewrites (CSN prompt set only).

Ye et al. Design Space Analysis As Ye et al.'s approach is dependent on specific parameters including (i) number of times the code is truncated and rewritten; and (ii) model used for rewriting (note, this is different from the model used to generate the AI portion of the dataset), we investigate sensitivity of the results on those choices. Figure 6 displays the resulting F1 scores on our whole dataset for the CSN prompt set (we omit the IBM prompt set for brevity). Results suggest no clear trend and for all other experiments we pick the combination of parameters which maximizes the average of all measured metrics across both prompt sets (8 rewrites w/ GPT).

5.3 Factors affecting classification

Impact of Training and Evaluation Set For classifiers which requires training, a relevant question is whether mismatch between training and evaluation set can impact performance. The model used for generating samples to be classified may differ from the model used for generating training samples. Further, the nature of coding tasks may differ, which may result in code samples with different characteristics. To evaluate the joint impact of these factors, we proceed as follows. First, we define an experiment as a combination of three factors: the *Classifier* being used, the combination of *Problem sets* used for training/evaluation (e.g., *CSN_IBM*), and the combination of *Models* used for generating



(a) Effect of training/evaluation set on F1 score for GPTSniffer classifier

(b) Effect of training/evaluation set on F1 score for Bukhari et al. classifier

Fig. 7: Training/evaluation set characteristics and F1 scores

the training and evaluation set (e.g., *GPT_Gemini*). We run a code classification experiment on all feasible combinations of factors, recording Precision and Recall, and we tabulate the results. Finally, we build a random forest regressor predicting each metric from the factors, and we extract Gini feature importance.

We resort to this approach as structured statistical tests would require a large number of repeated samples for each experiment, which are prohibitive to obtain due to the large and costly amount of computation required for individual experiments. While we verified the variation among repetitions to be minimal, we prefer the regression approach as it does not require repeated measures. We acknowledge that this approach does not allow us to generalize results beyond our set of experiments, but we believe that it is sufficient to identify general trends and qualitatively identify significant factors that may affect classification. As our goal here is not to achieve generalization but to maximize explanatory power, we perform hyper-parameter tuning.

The Precision regressor achieves good explanatory power ($R^2 = 0.78$), ranking Problem sets as the top feature ($Gini = 0.84$), followed by Models (0.10) and Classifier (0.07). The Recall regressor only achieves $R^2 = 0.46$ but ranks factors in the same order, respectively, with Gini scores 0.54, 0.25, 0.21. These results suggest that, regardless of the algorithm being used, training-based classifier performance is largely defined by differences between the characteristics of code used for training and classification, with the AI model used for code generation also playing a role.

To further investigate this observation, we plot heatmaps depicting how F1 scores vary based on the combination of training and evaluation set characteristics (i.e., the combination of problem set and model used for the generation of AI samples). Results for both GPTSniffer and Bukhari et al. are shown in Figure 7. The plots clearly show how classifier performance degrades significantly when the training and evaluation sets differ, particularly in terms of the problem set.

Approach	GPT 4o			Gemini			Claude		
	P	R	F1	P	R	F1	P	R	F1
Bukhari	0.84±0.01	0.79±0.02	0.82±0.01	0.84±0.01	0.79±0.02	0.82±0.01	0.84±0.01	0.79±0.02	0.82±0.01
GPTSniffer	**0.92±0.01**	**0.99±0.00**	**0.96±0.00**	**0.92±0.01**	**0.99±0.00**	**0.96±0.00**	**0.92±0.01**	**0.99±0.00**	**0.96±0.00**
Ye et al.	0.67±0.04	**0.61±0.07**	**0.63±0.04**	0.67±0.04	**0.61±0.07**	**0.63±0.04**	0.67±0.04	**0.61±0.07**	**0.63±0.04**
DetCodeGPT	**0.52±0.04**	0.95±0.08	0.67±0.01	**0.52±0.04**	0.95±0.08	0.67±0.01	**0.52±0.04**	0.95±0.08	0.67±0.01

Table 4: Base Prompt Variation Results: Each row presents the mean ± standard deviation for Precision, Recall, and F1 score across Base Prompts 1-5, showing the impact of different prompts on classifier performance. Highest values in green, lowest values in purple.

Approach	GPT+Gem vs Cl			GPT+Cl vs Gem			Gem+Cl vs GPT		
	P	R	F1	P	R	F1	P	R	F1
Bukhari et al.	0.96	0.90	0.93	0.93	0.82	0.87	0.90	0.59	0.71
GPTSniffer	0.95	0.66	0.78	0.91	0.92	0.91	0.96	0.73	0.83

Table 5: Multi-model, multi-problem training set: Each row presents Precision, Recall, and F1 score for a classifier trained on two out of three models, using both CSN and IBM problem sets.

Impact of Prompt Template. Each prompt used in our AI code generation experiments consists of two main components. The first part is the base prompt, which provides the model with instructions on how to generate code, what aspects to focus on, and the required output format. The second part is a detailed problem description from either the CSN or IBM dataset. During the code generation process, we kept the base prompt constant while iterating through problem descriptions from the datasets.

In this section, we investigate whether the choice of prompt template used for generation affects classification results. Our base prompt is discussed in Section 3.3 and given in Figure 3. In this experiment, we designed five additional base prompts to assess their impact on classification performance. To design of the variations, we performed an extensive literature review [4, 7, 15, 18, 42] and syncretized five styles: *Validation-Centric*, *Minimalist*, *Self-Contained*, *Strict Output-Only*, and *Testing-Oriented*. Content and description of each style are provided in our data package (see Section 1). We evaluate the effect of each prompt template by randomly selecting 100 problems (evenly distributed among the CSN and IBM sets) and feeding them to each of the models used for generation, resulting in a dataset of 300 instances (we used 4-fold cross-validation for training-based classifiers, and we fed the whole dataset to zero-shot classifiers).

Results are presented in Table 4. For brevity's sake, for each prompt/model combination we list the average value for each metric and the standard deviation. Variations across different base prompts are relatively small, which suggests that the choice of base prompt has a minimal effect on the final generated code and, consequently, on the classification process.

Approach	CSN vs IBM			IBM vs CSN		
	P	R	F1	P	R	F1
Bukhari et al.	0.63	0.43	0.51	0.27	0.84	0.41
GPTSniffer	0.50	0.99	0.67	0.51	0.99	0.67

Table 6: Multi-model, single-problem training set: each row presents Precision, Recall and F1 score for a classifier trained on samples generated by all models on one problem set, and evaluated on samples generated by all models for a different problem set.

5.4 Impact of diversified training set

For models that require training, another relevant question is whether diversification aids classification performance. In particular, we ask: (i) does diversifying the training set helps with classifying samples from previously unseen models?; and (ii) does diversifying the training set helps with classifying samples from a previously unseen problem set?

To answer the first question, we run one-out experiments where classifiers are trained using combined samples from CSN and IBM generated by two models and evaluated on samples generated by the third (we split the problem set so each problem only appears in either training or evaluation). Results are shown in Table 5. These results suggest that diversifying the training set does not consistently improve detection performance on unseen models. For example, Bukhari et al. suffers from low recall (0.28 average) in detecting Gemini/CSN when trained on GPT/CSN or Claude/CSN only (see Figure 7(b)). When trained on GPT+Claude/CSN+IBM, its recall jumps to 0.87. However, the same classifier exhibits an average recall of 0.71 in detecting GPT/CSN based on training on Gemini/CSN or Claude/CSN, while in this experiment, its recall falls to 0.59.

To answer the second question, we run four experiments where (i) we train each classifier on all samples generated from all models on the CSN problem set, and evaluate them on the IBM problem set; and (ii) vice versa. Results are presented in Table 6. Metrics in columns 2-4 were computed by training on the CSN problem set and evaluating on the IBM problem set; metrics in columns 5-7 were computed by training on IBM and evaluating on CSN. Results remain poor, suggesting that diversifying the dataset in terms of the generator model does not enable the classifiers to generalize across problem domains.

Overall, results show that the benefits of diversifying the training dataset may be limited, at least for the current generation of AI code detectors. Classifiers are able, to an extent, to generalize even from training data generated by a single model to detect code from another model - as long as the problem set remains the same. However, they are unable to transfer insights gleaned from one set of software development tasks, to a different set of tasks. We further discuss these observations in Section 6.

6 Discussion

Threats to Validity. We mitigate threats to internal validity by explicitly modeling confounding factors such as model used for generation, problem domain, and

discrepancies between training and evaluation set. We strive to mitigate external validity threats by diversifying our set of models used for generation, and for considering two different coding problem datasets. We believe our selection of models to be representative; the popularity of GPT, Gemini and Claude is empirically confirmed by the fact that these are the three models supported by the popular GitHub Copilot plugin [9]. While our selection of problem sets is limited to CSN and IBM, results show that they are sufficient to identify limitations of existing models, and they both consist of high-quality prompts with directly or indirectly vetted human solutions. As for construct validity, some models are sensitive to design parameters. We pre-analyze the effect of such parameters to ensure each model is tested under the most favourable conditions.

Another limitation of our study lies in the scope of the analyzed code sample: we focused exclusively on comparing source code that was entirely authored by humans with source code that was entirely generated by AI. This binary distinction enabled clearer ground-truth labeling and more controlled experiments. However, it does not capture the scenario in which human-authored and AI-generated code is intertwined—such as when developers use AI assistants to suggest, complete, or modify code—highlighting the need for future work to study these mixed-authorship settings. Thus, our current approach is inherently geared towards a best-case classification scenario, where the inputs represent fully human or fully AI-generated code.

A final limitation is that our study exclusively focused on Python source code. While this decision removes language as confounding factor, it nonetheless restricts the generalizability of our findings to other programming languages. Characteristics such as syntactical, lexical, and style may vary significantly across languages and may influence the performance of code origin classifiers. As a result, further investigation is necessary to evaluate whether our methods and conclusions hold when applied to code written in other coding languages.

Generated Code Functionality Preservation. An orthogonal but relevant question is whether AI-generated code correctly implements the desired functionality (human code samples in our datasets are directly or indirectly pre-vetted for correctness). In a small-scale experiment evaluating the correctness of machine-generated code, we compared 25 human-written code samples from the IBM problem set with AI outputs from the three models under examination. The IBM problems typically include sample test inputs/outputs, which we used to generate per-problem unit tests. Claude achieved a 48% pass rate (12/25), Gemini 24% (6/25), while GPT attained 52% (13/25). Notably, all three models succeeded concurrently in only 24% of cases (6/25), whereas 40% (10/25) of samples failed across the board, often due to recurring issues such as timeouts, incorrect values, and missed edge cases. In these cases, human programmers are likely not to use AI-generated code "as is", but rather modify it to correct recurring errors, resulting in mixed-authorship samples discussed above.

Implications and Future Directions. Results presented in Section 5 present a complex picture, with some clearly identifiable insights. First, zero-shot models

tend to perform poorly, with precision oftentimes below 70%, and recall below 80%. Models based on training or fine-tuning can achieve high performance, provided that the code submitted for classification comes from the same problem domain (i.e., problem set) used for training/fine-tuning. This observation is consistent with past work [22], and with the observation that human- and AI-generated code for the same problem set presents distinct statistical properties (ref. Section 3.3). The same-origin assumption may be reasonable in some contexts – for example, detecting AI code in implementations of well-defined critical functionality, such as implementations of specific ciphers.

In general, however, existing tools do not appear to be able to detect AI-generated code in the wild, under realistic conditions, with sufficient accuracy to be practical. Thus, it may be worth investigating alternative solutions such as watermarking [33]. In situations where compliance requirements can be enforced, it may also be possible to use developer tools to track the use of AI code assistants and tag AI-generated code as it is created [5].

In parallel, incorporating developer perspectives may prove invaluable. This would entail conducting a surveys on the experiences of developers with AI detection tools, developer views on the performance and security of machine-generated code, and the broader culture surrounding the use of AI-assisted programming.

7 Related Work

Supply Chain Security. Supply chain security issues emerge from current software design practices where code from various, potentially untrusted origins is incorporated into a project. This increases the risk of malicious, vulnerable, or other undesirable code being present in projects. Recent work includes identifying malicious packages, such as in PyPI [10] or npm [1]. Such solutions are, however, imperfect, suggesting that any externally sourced code should be tracked and examined. As AI-generated code potentially introduces new threat vectors into the supply chain (e.g., through the generation of insecure code), there is a need for automated processes that can complement existing human-driven software practices for provenance tracking and analysis; our work provides insight into the current state-of-the-art on this front.

Code Classification and Stylometry. Author identification using coding style dates back to the 1980s, with Oman et al. [23] pioneering the approach by analyzing the typographic and layout styles of programs to identify the authors of three Pascal algorithms from various computer science textbooks. More recently, Caliskan et al. [6] developed the Code Stylometry Feature Set (CSFS) for programmer de-anonymization, specifically for identifying human authors from a set of potential candidates. Caliskan's technique, used for source code attribution, is considered a closed-world machine learning task involving multi-class classification. CSFS offers a comprehensive code representation, categorized into three feature types: lexical features, which reflect programmer choices like keyword, identifier, and operator frequency; layout features, capturing the visual structure

of the code, including indentation, line length, and comments; and syntactic features, derived from Abstract Syntax Trees (ASTs), which delve deeper into the code's structure by examining element types, nesting levels, and control flow.

Building upon the foundation laid by Caliskan et al.'s [6] Code Stylometry Feature Set (CSFS), researchers have explored various applications and extensions. Watson [37], for example, presented a method to de-anonymize source code contributors based on intrinsic programming style, building upon Caliskan-Islam et al.'s work but modifying the feature set and modeling strategy for improved scalability and feature-selection robustness. In our previous work [5], on the other hand, we have leveraged the CSFS for a binary classification task, distinguishing between AI-generated and human-written code. Nguyen et al. [22] developed GPTSniffer, a machine learning solution designed to detect source code potentially generated by ChatGPT. GPTSniffer's classification engine utilizes CodeBERT, a pre-trained model specialized in code analysis and trained on the extensive CodeSearchNet dataset. Shi et al. [32] introduced DetectCodeGPT, a novel method for identifying machine-authored code. DetectCodeGPT modifies code and analyzes the responses of a pre-trained model. It focuses on stylistic tokens like whitespaces and newlines. By strategically inserting such tokens, DetectCodeGPT aims to highlight stylistic differences between human and machine-written code, making identification easier. Ye et al. [41] developed a zero-shot method for detecting synthetic (AI-generated) code. Their method is based on the principle that the similarity between the original code and versions rewritten by LLMs is indicative of whether the code is AI-generated. The process involves rewriting the code and subsequently comparing the original and rewritten versions for similarity.

8 Conclusion

We performed a comparative investigation of the performance of tools for detecting AI-generated code. We investigated multiple classifiers, taking into account both the impact of the model used for generating code, and of different programming tasks. Results suggest that classifiers can be effective under narrow assumptions, but are not yet sufficiently accurate to be used in the wild.

Acknowledgements: We thank the anonymous reviewers for their feedback. We also thank Brian Meta, Masroor Posh and Elizabeth Wyss for their help with this work. This work was supported by NSERC Alliance Grant #2341206 "Managing Risks of AI-generated Code in the Software Supply Chain".

References

1. Adriana Sejfia, Max Schafer: Practical Automated Detection of Malicious npm Packages. In: ICSE (2022)
2. Anthropic: Introducing claude. `https://www.anthropic.com/news/introducing-claude` (Mar 2023)

3. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program synthesis with large language models (2021), https://arxiv.org/abs/2108.07732
4. Brown, T.B., et al.: Language models are few-shot learners (2020), https://arxiv.org/abs/2005.14165
5. Bukhari, S., Tan, B., De Carli, L.: Distinguishing AI- and Human-Generated Code: A Case Study. In: ACM CCS SCORED Workshop (2023)
6. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R.: De-anonymizing Programmers via Code Stylometry. In: USENIX Security Symposium (2015)
7. Chen, M., et al.: Evaluating large language models trained on code (Jul 2021), http://arxiv.org/abs/2107.03374
8. Claburn, T.: GitHub and OpenAI fail to wriggle out of Copilot lawsuit. https://www.theregister.com/2023/05/12/github_microsoft_openai_copilot/ (May 2023)
9. Dohmke, T.: Bringing developer choice to copilot with anthropic's claude 3.5 sonnet, google's gemini 1.5 pro, and openai's o1-preview. https://github.blog/news-insights/product-news/bringing-developer-choice-to-copilot/ (Oct 2024)
10. Duc Ly Vu, Zachary Newman, John Speed Meyers: Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In: ICSE (2023)
11. Friedman, N.: Introducing GitHub Copilot: Your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/ (Jun 2021)
12. GPTZero: AI Detector - the Original AI Checker for ChatGPT & More. https://gptzero.me/
13. HackerNews: Ask HN: Does your company ban GitHub Copilot? | Hacker News. https://news.ycombinator.com/item?id=34914810 (Feb 2023)
14. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., Steinhardt, J.: Measuring coding challenge competence with apps (2021), https://arxiv.org/abs/2105.09938
15. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search (2020), https://arxiv.org/abs/1909.09436
16. IBM Research: github/project_codenet. https://github.com/IBM/Project_CodeNet (Jan 2025)
17. Idialu, O.J., Mathews, N.S., Maipradit, R., Atlee, J.M., Nagappan, M.: Whodunit: Classifying Code as Human Authored or GPT-4 Generated – A case study on CodeChef problems. In: MSR (2024)
18. Mathews, N.S., Nagappan, M.: Test-driven development and llm-based code generation. In: ASE (2024)
19. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **SE-2**(4) (Dec 1976)
20. Meta AI: Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/ (2025)
21. Mitchell, E., Lee, Y., Khazatsky, A., Manning, C.D., Finn, C.: Detectgpt: Zero-shot machine-generated text detection using probability curvature (2023), https://arxiv.org/abs/2301.11305
22. Nguyen, P.T., Di Rocco, J., Di Sipio, C., Rubei, R., Di Ruscio, D., Di Penta, M.: Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. Journal of Systems and Software p. 112059 (2024)

23. Oman, P.W., Cook, C.R.: Programming style authorship analysis. In: CSC (1989)
24. OpenAI: Introducing ChatGPT. https://openai.com/index/chatgpt/ (Nov 2022)
25. OpenAI: Best practices for prompt engineering with the OpenAI API | OpenAI Help Center. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api (2025)
26. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In: IEEE S&P (2022)
27. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do users write more insecure code with ai assistants? In: ACM CCS (2023)
28. Pichai, S., Hassabis, D.: Introducing Gemini: Our largest and most capable AI model. https://blog.google/technology/ai/google-gemini-ai/ (Dec 2023)
29. Roberto Torres: Apple restricts ChatGPT, GitHub Copilot use over data worries: Report. https://www.ciodive.com/news/apple-chatgpt-openai-copilot-generative-AI/650816/ (May 2023)
30. Schuster, R., Song, C., Tromer, E., Shmatikov, V.: You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In: USENIX Security Symposium (2021)
31. Shani, I., GitHub Staff: Survey reveals AI's impact on the developer experience. https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/ (Jun 2023)
32. Shi, Y., Zhang, H., Wan, C., Gu, X.: Between lines of code: Unraveling the distinct patterns of machine and human programmers. In: ICSE (2025)
33. Suresh, T., Ugare, S., Singh, G., Misailovic, S.: Is the watermarking of llm-generated code robust? (2025), https://arxiv.org/abs/2403.17983
34. TIOBE: Tiobe index. https://www.tiobe.com/tiobe-index/ (2025)
35. Tufano, R., Mastropaolo, A., Pepe, F., Dabić, O., Di Penta, M., Bavota, G.: Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In: MSR (2024)
36. Vaidya, R.K., De Carli, L., Davidson, D., Rastogi, V.: Security Issues in Language-based Software Ecosystems. http://arxiv.org/abs/1903.02613 (Nov 2021)
37. Watson, D.: Source Code Stylometry and Authorship Attribution for Open Source. Ph.D. thesis, University of Waterloo (Sep 2019)
38. Weidinger, L., Uesato, J., Rauh, M., Griffin, C., Huang, P.S., Mellor, J., Glaese, A., Cheng, M., Balle, B., Kasirzadeh, A., Biles, C., Brown, S., Kenton, Z., Hawkins, W., Stepleton, T., Birhane, A., Hendricks, L.A., Rimell, L., Isaac, W., Haas, J., Legassick, S., Irving, G., Gabriel, I.: Taxonomy of Risks posed by Language Models. In: ACM FAccT (2022)
39. Yan, S., Wang, S., Duan, Y., Hong, H., Lee, K., Kim, D., Hong, Y.: An LLM-Assisted Easy-to-Trigger Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities against Strong Detection. In: USENIX Security Symposium (2024)
40. Yang, X., Zhang, K., Chen, H., Petzold, L., Wang, W.Y., Cheng, W.: Zero-shot detection of machine-generated codes. https://arxiv.org/abs/2310.05103 (2023)
41. Ye, T., Du, Y., Ma, T., Wu, L., Zhang, X., Ji, S., Wang, W.: Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. https://arxiv.org/abs/2405.16133 (2024)
42. Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J.: Large language models are human-level prompt engineers (2023), https://arxiv.org/abs/2211.01910