An Empirical Study to Evaluate AIGC Detectors on Code Content

Jian Wang[*]
Singapore Management University
Singapore
jwang@smu.edu.sg

Shangqing Liu[†]
Nanyang Technological University
Singapore
liu.shangqing@ntu.edu.sg

Xiaofei Xie
Singapore Management University
Singapore
xfxie@smu.edu.sg

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

ABSTRACT

Artificial Intelligence Generated Content (AIGC) has garnered considerable attention for its impressive performance, with Large Language Models (LLMs), like ChatGPT, emerging as a leading AIGC model that produces high-quality responses across various applications, including software development and maintenance. Despite its potential, the misuse of LLMs, especially in security and safety-critical domains, such as academic integrity and answering questions on Stack Overflow, poses significant concerns. Numerous AIGC detectors have been developed and evaluated on natural language data. However, their performance on code-related content generated by LLMs remains unexplored.

To fill this gap, in this paper, we present an empirical study evaluating existing AIGC detectors in the software domain. We select three state-of-the-art LLMs, i.e., GPT-3.5, WizardCoder and CodeLlama, for machine-content generation. We further created a comprehensive dataset including **2.23M** samples comprising code-related content for each model, encompassing popular software activities like Q&A (**150K**), code summarization (**1M**), and code generation (**1.1M**). We evaluated **thirteen** AIGC detectors, comprising **six** commercial and **seven** open-source solutions, assessing their performance on this dataset. Our results indicate that AIGC detectors perform less on code-related data than natural language data. Fine-tuning can enhance detector performance, especially for content within the same domain; but generalization remains a challenge.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Security and privacy** → *Social aspects of security and privacy*; • **Computing methodologies** → **Natural language generation**.

[*]Also with Nanyang Technological University.
[†]Shangqing Liu is the corresponding author.

KEYWORDS

AIGC Detection, Code Generation, Large Language Model

ACM Reference Format:
Jian Wang, Shangqing Liu, Xiaofei Xie, and Yi Li. 2024. An Empirical Study to Evaluate AIGC Detectors on Code Content. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695468

1 INTRODUCTION

In recent years, Artificial Intelligence Generated Content (AIGC) has attracted significant attention and interest from academia and industry. AIGC refers to content that is generated by advanced generative AI techniques. With AI techniques becoming more advanced, the generated content shows significantly better quality and is being used in a wide range of tasks. ChatGPT [5], released by OpenAI, has become one of the most attention-grabbing approaches. Followed by ChatGPT, a series of large language models have been proposed, such as Llama2 [66], GPT-4 [20].

Large language models have demonstrated remarkable proficiency in generating content across a diverse range of domains. Their abilities to comprehend context, adhere to instructions, and produce coherent content make them particularly well-suited for tasks such as drafting emails, generating articles, composing poetry, crafting stories, and producing social media content. Furthermore, they have confirmed their capabilities in software development and have been widely used in software development tasks such as writing documentation, creating user manuals, generating code snippets, reviewing code and repairing code.

Although LLMs offer numerous benefits for users, it is important to consider the potential for abuse. In the educational domain, for instance, there is a risk that students may use LLMs to cheat on exams or plagiarize assignments, which violates academic integrity. To avoid abuses, some universities have restricted the use of LLMs, as shown in the recent report[1]. Similarly, in the industry, the source of content generated by LLMs must be carefully considered, especially in security and safety-critical scenarios. AI-generated content may have low quality or contain errors (e.g., toxic content or bugs) that could lead to serious consequences [23]. For example, to prevent malicious use of the contents generated by ChatGPT when answering questions, Stack Overflow has announced that the

[1]https://www.universityworldnews.com/post.php?story=20230222132357841

ChatGPT-generated content is temporarily banned,[2] because "*the average rate of getting correct answers from ChatGPT is too low, the posting of answers created by ChatGPT is substantially harmful to the site and to users who are asking and looking for correct answers*".

With the increasing use of LLMs in a wide range of domains, including software development, it becomes more crucial to develop effective tools to detect AI-generated content. For example, many AIGC detectors [1, 2, 4, 6, 9, 11, 12, 16, 18, 19, 31, 56] from both academia and industry has been developed to detect the generated contents from GPT-series models, including GPT-2 [60], GPT-3 [25], and ChatGPT. While these tools have been proposed to detect LLM-generated content, it remains unclear how effective these tools are, particularly in the context of the software development domains. Most existing detection tools are evaluated on natural language inputs, but it is still unknown whether they can also detect LLM-generated code effectively.

To fill this gap, in this paper, we take a further step and conduct a comprehensive empirical study to evaluate the existing detectors, including both the open-source and commercial ones, on their capacities of detecting the code-related content (e.g., code and documents) generated by LLMs. Specifically, the study aims to answer crucial questions as follows: *How accurate are the current tools for detecting code-related content generated by LLMs? What are the differences in performance between detecting natural language content and code-related content generated by LLMs? Can fine-tuning the detection tools enhance their capability to identify ChatGPT-generated content? How robust are the detection tools in detecting content that has been modified based on LLM-generated content?*

To conduct this study and answer this question, We use three state-of-the-art LLMs, namely GPT-3.5-Turbo, WizardCoder-15B, and CodeLlama-34B-Instruct [3] for machine-content generation. For each model, we constructed two datasets, namely the Code-Related Content Dataset (*CCD*) and the Natural Language-Related Content Dataset (*NLCD*), by generating related content using LLMs in the domains of programming and natural language, respectively. *CCD* consists of **1.08M** samples across text-to-code scenario including three code generation datasets APPS, CONCODE and Doc2Code-LLM. *NLCD* consists of **1.16M** samples across Q&A from stack overflow and code-to-text generation [39]. Note that each sample in *CCD* and *NLCD* is a pair, including the human-generated data and LLM-generated data.

Based on this dataset, we design comprehensive experiments to evaluate the capabilities of existing detection tools, including **seven** open-source detectors and **six** commercial detectors. We evaluate the performance of selected tools in detecting program contents generated by LLMs with those generated by human. Extensive experiments have revealed that current AIGC detectors struggle to detect code-related data compared to natural language data. Although fine-tuning is able to improve performance, however, the generalization capacities are limited. Overall, the main contributions of our paper are summarized as follows:

- We conducted a comprehensive empirical study to evaluate the performance of **thirteen** AIGC detectors, including **seven** open-source detectors and **six** commercial detectors, on detecting code-related content generated by GPT-3.5, WizardCoder and CodeLlama.
- We construct two large-scale datasets namely *CCD* and *NLCD*, consisting of **1.08M** code-related samples and **1.16M** natural language-related samples. We have made our code and data public[4] to facilitate the following research.
- The extensive experiments have indicated that AIGC detectors perform worse on code-related data. Fine-tuning can enhance the detector performance but is still limited to generalization.

2 EXISTING DETECTORS

To ensure the responsible and ethical use of AI-generated content, various detectors have been developed to identify whether a given piece of content was generated by an AI model. We have collected multiple detectors [1, 2, 4, 6, 9, 11, 12, 16, 18, 19, 31, 56] as up to 2024, and their detailed information is presented in Table 1. The "Detectable Models" column lists the types of models supported by the detectors, where "Unknown" means the supported model is unclear from the official documentation. Column "Interfaces" indicates the supported interfaces through which the detection tools may be accessed from. For example, "Website" denotes that the detector can only be accessed from its official website, and "API" means it supports access from standard programming interfaces. It is worth noting that some commercial detectors that support API access may not be free or may only allow a limited number of visits per day. For instance, Sapling [16] provides API access at a cost of 25 dollars per month and Writefull [18] restricts access when the daily quota is reached. For instance, GPTZero.me [12] provides API access at a cost of 35 dollars per month and restricts access when the daily quota is reached. Our collaborators manually registered dozens of accounts for supporting our experiments. Finally, as is shown in the "Input Length" column, each detector may have a different requirement on the length of the input texts. For example, the open-source detectors can only process input texts up to 512 tokens. The input lengths for the commercial detectors are also different, which are presented in Table 1.

3 STUDY DESIGN

In this section, we give details on our study design. Our study is centered around typical scenarios where LLMs have been used to support software development activities. We collected data from both LLMs and human experts in each usage scenario and then compared the performance of different detectors.

3.1 Scenarios and Data Collection

LLMs have been widely used in software development activities. For example, they can be used to answer programming-related questions, summarize code snippets with natural languages, and generate code based on natural language descriptions. In this study, we select three state-of-the-art large language models, including GPT-3.5-Turbo, WizardCoder-15B and CodeLlama-34B-Instruct, for study. The selected models have shown their superiority over other

[2]https://meta.stackoverflow.com/questions/421831/temporary-policy-chatgpt-is-banned

[3]Note that these models have confirmed their superiority over other LLMs during our project. We cannot include GPT-4 for experiments due to the limited budget.

[4]https://sites.google.com/view/nlccd

Table 1: The details of the existing AIGC detectors.

Detector		Detectable Models					Interfaces		Input Length
		GPT-2	GPT-3	ChatGPT	Llama	Unknown	Website	API	
	GPT2-Detector [1]	✓						✓	<512 tokens
	DetectGPT [56]	✓					✓	✓	<512 tokens
	RoBERTa-QA [32]			✓			✓	✓	<512 tokens
Open-source	ArguGPT [53]	✓	✓	✓			✓	✓	<512 tokens
	MAGE [45]	✓	✓	✓			✓	✓	<2048 tokens
	RADAR [38]	✓	✓	✓			✓	✓	<512 tokens
	FastDetectGPT [24]	✓	✓	✓				✓	<512 tokens
	Multiscale [65]	✓	✓	✓			✓	✓	<512 tokens
	Grover.Allenai [13]	✓					✓	✓	100 chars to 100k chars
	Compilatio.net [6]		✓	✓			✓		200 chars to 2k chars
	Contentatscale.ai [2]		✓	✓			✓		25 words to 25k chars
	CopyLeaks.com [4]		✓	✓			✓	✓	150 chars to 25k chars
	Crossplag.com [14]			✓			✓	✓	<3k words
Commercial	GPTZero.me [12]	✓	✓	✓	✓		✓	✓	250 chars to 5k chars
	Originality.ai [15]			✓	✓		✓		50 words to 300words
	Sapling.ai [16]		✓	✓	✓		✓	✓	50 words to 2k chars
	Scribbr.com [11]	✓	✓	✓			✓	✓	25 words to 500 words
	Writefull.com [18]		✓	✓			✓	✓	50 words to 2k words
	Writer.com [19]					✓	✓	✓	<1.5k chars

LLMs. We exclude GPT-4 as its expensive cost. Furthermore, we focus on three of the most common scenarios in software development: (1) Q&A on programming topics, (2) code-to-text generation, and (3) text-to-code generation. To conduct our study, we first collect relevant data from both humans and LLMs. Then, we evaluate the capacity of different detectors to detect LLM-generated contents in these scenarios.

Data Filtering. Through our careful inspection of the generated LLMs' results from the preliminary experiments, we find that some of them have identifiable symbols or phrases indicating that they are generated by LLM. For example, some examples may have evident clues such as "As a language model..." to indicate it is generated by LLM. Thus, we need to remove these samples to ensure the constructed dataset is non-trivial. Formally, we defined the following rules for filtering. Firstly, if we find the generated content has the keywords 'ai model', 'artificial intelligence', or 'language model' (case-insensitive), we remove these samples. Secondly, we filter out samples that come with sentences such as, *'I apologize for the confusion, but I am unable to ....'*. More such responses are provided on our website.[5]4 Thirdly, for the code generated by LLMs, we extract the code block marked by three backticks from LLM and then remove the comments from the code block to only keep the code contents. With these steps, we try to enhance the quality of the constructed dataset and apply these steps to the subsequent data collection process.

3.1.1 *Q&A.*

It is a common practice for programmers to search the Internet for answers when they have questions on certain programming tasks. Q&A websites, such as Stack Overflow, are designed for this purpose. Stack Overflow collects and organizes relevant answers, which become an essential resource for software developers today. Stack Overflow expects high-quality answers from genuine experts to build a healthy and sustainable community (as indicated in its current policy). Therefore, effective detection of AI-generated contents with high accuracy is necessary to support a sustainable Q&A service. Our first scenario focuses on studying the effectiveness of AIGC detectors in identifying programming-related answers generated by LLMs.

Data Collection. To evaluate the effectiveness of AIGC detectors in the Q&A scenario, we used the Stack Overflow dataset from Stack Exchange [7] which includes questions and answers posted on the Stack Overflow platform from September 2021 to November 2022. We select the questions from SE-related domains, such as code reviews, competitive programming challenges, data science and machine learning questions, web development, and applications, etc. For each question, we follow the standard process [3] to select the answer that is accepted by the author as the human-generated answer and use LLM to generate another answer in response to the same question. We follow the following instructions [10] for the design of the prompt:

> *I want you to act as a Stackoverflow post. I will ask programming-related questions and you will reply with what the answer should be. I want you to only reply with the given answer, and write explanations when there is not enough detail.*
> *<Question>*

where <Question> is the question we would like LLMs to answer. In total, after the data filtering step, for each selected model, we obtained **150K** pairs of human-generated and LLM-generated answers for the **150K** questions. This dataset (denoted as Q&A-LLM) provides a comprehensive benchmark for evaluating the performance of detectors in identifying LLM-generated content in the context of programming-related questions and answers.

3.1.2 *Code-to-Text Generation.*

Generating natural language descriptions of a given code snippet has been a long-standing research challenge widely studied in academia [21, 40, 50]. Accurate descriptions of code can help programmers better understand its functionality and improve software development efficiency. However, writing accurate code descriptions is a time-consuming and laborious task. LLM has demonstrated excellent capabilities in generating natural language descriptions of code, making it a promising solution for automating this task. Yet, it is still not advisable to replace human-written code descriptions with machine-generated code descriptions, for the lack of quality guarantees. Therefore, detectors that are capable of identifying code descriptions generated by LLM are necessary to discover massive use of machine-generated code descriptions.

Data Collection. Specifically, we adopted the widely used benchmark CodeSearchNet [39], where each sample is a pair (*code*, *description*), across six programming languages including Ruby, Javascript, Go, Python, Java, and PHP. Each sample has a *docstring* field which contains the descriptions of the code produced by human experts. To obtain code descriptions from LLMs, after several attempts to verify the quality of the generated content through different prompts, we select a better prompt and ask LLMs to generate the summary of the given code. The prompt is as follows:

> *You will be given a <LANG> function code and your task is to generate a detailed summary of its behavior and functionality. Your summary should clearly explain what the function does, how it works, and what input parameters and output values it expects. You should write your explanation in clear and concise language.*
> *<CODE>*

where <LANG> is one of the six programming languages, and <CODE> is the target code we would like to summarize its description. In total, for each model, we obtain **1M** samples from CodeSearchNet and generate the answers using LLM, denoted as Code2Doc-LLM. Each programming language accounts for a different number of samples. The number of Go, Java, Javascript, PHP, Python, and Ruby is 182K, 181K, 65K, 268K, 281K and 27K respectively. This dataset provides a comprehensive benchmark for evaluating the performance of detectors in identifying natural language descriptions of code generated by LLM across a range of programming languages.

In the NL generation, we generated a total of **1.15M** pairs of natural language samples, where each pair consists of human-generated and LLM-generated natural language content.

3.1.3 Text-to-code Generation. With the advancements in AI technology, particularly the development of large language models, there has been a surge of interest in automatically generating code from natural language descriptions. Recent works [26, 58] have revealed that LLMs are powerful at writing programs following human instructions. The use of AI-generated code may be prohibited in some contexts, for example, due to policies on academic integrity. Our third scenario studies the effectiveness of the existing detectors, to better understand the technical feasibility of distinguishing code generated by LLM from those written by human.

Data Collection. Specifically, we collected the LLM-generated code based on three different code generation datasets, i.e., APPS [36], CONCODE [41], and the Code2Doc-LLM dataset described in Section 3.1.2.

- The APPS dataset [36] is a Python dataset consisting of coding problems gathered from various public websites. Each problem in the dataset is accompanied by its description, ground-truth solutions, and test cases used to validate the implemented solutions. We regard the ground-truth solutions as the answers provided by human experts. To obtain solutions from LLM, we design the prompt as follows:

  Please complete the Python code generation for the following question, there may be some example test cases, and you can use them to evaluate the generated code. Do not provide any explanations, comments, test cases, or additional text, only output the completed Python code in a markdown style and nothing else.
  <Question>

  where <Question> is the placeholder of the problem description and the function name provided in APPS. We finally collected **8.7K** samples for each model, denoted as APPS-LLM dataset.
- CONCODE [41] is a Java dataset included in the CodeXGLUE [54] collection. Its goal is to generate class member functions for a Java class based on natural language descriptions and the programmatic context provided by the class environment, which includes member variables and other member functions in the class. To obtain answers from LLM, the prompt is designed following the task description from CodeXGLUE:

  Generate the source code of class member functions in Java, given natural language description and class environment. The class environment is the programmatic context provided by the rest of the class, including other member variables and member functions in the class. Please only reply with a code block and avoid providing any explanations, comments, imports, or additional text.
  The nature language description is <Description>, the member variables and member functions is <Class>.

  where <Description> and <Class> are the placeholders of the description and the programmatic context provided by the rest of the class, including other member variables and member functions in the class. The ground truth code provided in CONCODE is considered the answer from human experts. We collected a total of **66K** samples for each selected model, which we refer to as CONCODE-LLM dataset.
- The Code2Doc-LLM dataset constructed in Section 3.1.2 consists of detailed descriptions of given code generated by LLM. We can naturally ask LLM to generate code based on the code descriptions generated by LLM in the Code2Doc-LLM dataset. To do so, we designed a prompt for generating code, as shown follows:

  You will be provided with a detailed description of a function, and your task is to generate a function that implements the program's behavior based on that description. You should write the function code as accurately as possible based on the description, without providing any additional explanations or assumptions. Your implementation should conform to the standard of syntax and coding conventions.,

  The original code in the Code2Doc-LLM dataset is considered to be the data from human experts. Additionally, we collected a total of **1M** code samples generated by LLMs, which we refer to as the Doc2Code-LLM dataset.

In the code generation, for each model, we generated a total of **1.08M** pairs of code samples, where each pair consists of human-generated and LLM-generated code.

Table 2: The statistics of the collected data for each selected large language model.

Split	NLCD		CCD		
	Q&A-LLM	Code2Doc-LLM	Doc2Code-LLM	CONCODE-LLM	APPS-LLM
Train	148K	908K	904K	65K	5K
Test	2K	97K	96.6K	1.4K	3.7K
Total	150K	1M	1M	66K	8.7K

In summary, we collected **2.23M** samples for each selected large language model. As the original CodeSearchNet provides the train-valid-test split, we can directly use it to get the data split for the LLM-generated dataset. We combine the validation and test sets of Doc2Code-LLM, and Code2Doc-LLM respectively to expand the scale of the testing for detectors. In addition, CONCODE does not provide the groud-truth for the testset, hence we just use the validation set for testing. Furthermore, as the data split of Q&A dataset has not been provided, we randomly select 2k samples for testing and the remaining 148k samples for training. The detailed statistics of the NLCD and CCD are present in Table 2.

3.2 Selected Detectors

In this section, we introduce the AIGC detectors selected for this study. The detectors are divided into two categories: commercial and open-source. We chose **six** commercial detectors, namely Grover, Crossplag.com, Originality.ai, GPTZero.me, Scribbr.com, and Writer.com, as recommended by [17], due to their ability to perform large-scale testing efficiently in terms of time and cost. Additionally, to the best of our knowledge, there are currently **seven** open-source detectors, each utilizing different detection algorithms.

3.2.1 Commercial Detectors. We divide the selected commercial detectors into two sub-categories: free&unlimited quota and limited quota or pre-paid.

- Free & Unlimited Quota: ① Writer.com [19] is a free tool, though the technical details have not been revealed on its official website. However a small-scale study[6] indicated that it successfully detects 40% of human-written content and 80% of AI-written content. ② Scribbr.com [11] is a free AI detector, which is able to detect contents generated by GPT-2, GPT-3, ChatGPT, and GPT4. ③ Grover [13] was released by allenai.org in May 2019, focusing on defense against neural fake news. It employs a dual-modal system of generator and discriminator. They reported that Grover had achieved over 92% accuracy in distinguishing human-generated samples from those written by machines.
- Limited Quota or Pre-Paid : ① GPTZero.me [12], according to its official website, is an ensemble model that combines classification models with statistical approaches, providing predictions at the sentence, paragraph, and document levels. ② Crossplag.com [14] provides a variety of plagiarism detection tools and services aimed at helping educators and publishers uphold academic integrity and ensure the originality of their content. Their technical report mentions using strategies such as identifying inconsistencies, analyzing language patterns, and employing AI detectors to provide valuable insights. ③ Originality.ai [15], is trained on AI-generated content using a popular NLP model that includes data from GPT-4, ChatGPT, Claude-3, Gemini Pro 1.5, Llama, AI Paraphrase, and Humanizer. Notably, their new model, testing on GPT-4-Turbo, claims an accuracy of approximately 99% [8]

[6]https://www.bloggersgoto.com/writer-com-ai-content-detector-review

3.2.2 Open-source Detectors. Similarly, we categorize the open-source detectors into two sub-categories: supervised training on domain data and perturbation based training.

- Supervised Training on Domain Data: ① GPT2-Detector [1] is a fine-tuned detector from RoBERTa [52] released by OpenAI, with a training dataset derived from the outputs of the 1.5B-parameter GPT-2 model. ② RoBERTa-QA [31]. proposed by Guo et al., is trained on the Human ChatGPT Comparison Corpus (HC3) dataset, which includes 37K questions across financial, medical, legal, psychological, and open domains to detect content generated by ChatGPT. ③ ArguGPT [53] is a fine-tuned RoBERTa model designed to detect machine-generated argumentative essays. The training data is collected from essays such as TOEFL and GRE writing tasks, including 8k human-machine comparison essays (4k human vs. 4k machine). ④ MAGE [45] utilizes a **wild** dataset, collecting human-written texts from seven distinct writing tasks, including story generation, news writing, and scientific writing. It further utilizes 27 LLMs to generate AI-generated content for training their detection model.
- Perturbation-Based training: ① RADAR [38], aims to jointly train a robust AI-text detector via adversarial learning by simultaneously training a paraphraser and detector. Its performance significantly improves from 0.892 to 0.920 compared to DetectGPT [56] on the unseen dataset. ② MultiScale [65] introduces a length-sensitive Multiscale PU Loss for training, avoiding the need to truncate data forcefully by treating the training phase in multiple parts, cascaded with different scale factors. ③ FastDetectGPT [24] is an upgrade version for DetectGPT [56], substitutes DetectGPT's perturbation step with a more efficient method, the conditional probability function. This method offers not only substantial performance benefits over DetectGPT but is also much less compute intensive, their method accordingly uses a new criteria, the conditional probability curvature, which they find is more positive for LLM output than human. They report performance boosting with an average of 28%, and inference speedup 340 times.

3.3 Experimental Design and Research Questions

In this section, we will present the designed research questions and the experimental setup for each research question.

3.3.1 RQ1: How effective are existing detectors in detecting LLM-generated content? We will evaluate the performance of thirteen detectors on dataset *CCD-Test* and *NLCD-Test*.

Evaluation Setup: Specifically, if the input length of an answer exceeds the maximum sequence length required by the selected detector, we truncate it to meet the requirement. For some detectors, a threshold is needed to distinguish between LLM-generated data and human-generated data. An output probability greater than the defined threshold indicates that the content is generated by LLMs, such as ChatGPT. We adhere to the default settings of these detectors, where GPTZero is set to 0.8 and the other detectors are set to 0.5.

3.3.2 RQ 2: What factor of the generated content affects the detection performance? The first research question aims to explore the detection performance of current detectors across different code scenarios. We further want to explore the factor of the generated content that affects the detection performance. We investigate three crucial potential factors that might impact the detection performance, including code complexity, code length, and programming languages.

Due to the black-box nature of language models, it is unclear whether a given model will tend to generate simple or complex solutions, especially in coding scenarios where a problem may have multiple implementation solutions. Intuitively, the longer the content generated by the model, the more likely it is to expose certain latent features, enabling the detector to identify them. Additionally, since code can be implemented in different programming languages, the choice of programming language can also be a potential factor affecting detection performance.

Experimental Setup: We investigate the impact of code complexity, generated code length and different programming languages on the test set of Doc2Code-LLM.

For the code complexity analysis, We follow the previous work [62] to use the *cyclomatic complexity* algorithm, which measures the number of linearly independent paths through a program module and *a lower cyclomatic complexity is easier to understand this code* , to categorize each program into three types: Easy, Medium, and Hard. The criteria are based on the ratio between the number of linearly independent paths and the line of the code.

For the content length analysis, we first calculate the cut-off points to divide the entire dataset, which combines outputs from three LLMs, into three equal parts based on their tokenized length. The cut-off points are [0-72], [72-192], and [192+] in the entire Doc2Code-LLM-Test, noted Short, Medium, and Long, respectively.

Lastly, for the analysis of the effect of programming language, as Doc2Code-LLM consists of six programming data: Go, Java, Javascript, PHP, Python and Ruby. We directly use them for the analysis.

3.3.3 RQ3: To what extent can fine-tuning improve detection performance? Since the detectors we compared were primarily designed for detecting natural language content, they may not perform optimally on our code-related dataset. Therefore, in this question, we aim to investigate whether fine-tuning can enhance the performance of the detectors.

Experimental Setup: We selected the open-source detector RoBERTa-QA [31] for fine-tuning. Specifically, we fine-tuned seven detectors with the dataset *NLCD-Train* and *CCD-Train* in Table 2. The first three detectors fine-tuned with the training dataset of Q&A-LLM, Code2Doc-LLM and the composite of Q&A-LLM and Code2Doc-LLM, which are more related to detecting natural language data, while the remaining detectors fine-tuned on APPS-LLM, CONCODE-LLM, Doc2Code-LLM and the composite of APPS-LLM, CONCODE-LLM and Doc2Code-LLM, which are more related to code data. The fine-tuned detectors are evaluated on the same test dataset, i.e., *NLCD-Test* and *CCD-Test*.

3.3.4 RQ4: How robust are these detectors when the LLM-generated data is slightly modified? In real-world scenarios, the content generated by ChatGPT may not be used directly, and some of the generated content can be modified for customization to avoid detection. Similarly, for the human content that detectors can correctly identify, it is necessary to investigate whether this content can still be detected as human-written content after the modification. In summary, we aimed to investigate the robustness of detectors.

Experimental Setup: We use the detector that can correctly identify the human code and LLM-generated code from APPS-GPT for evaluation. We define five mutation operations to modify the Python code as follows:

- *FuncAddLine.* It aims to modify the function invocation to a different format. Specifically, the syntax for a function invocation in Python is *f(a, b, c)* where the parameter *a, b, c* is the formal parameter of the function *f*. We define the mutation operator to add the line breaks in the function for separation, for example changing *f(a, b, c)* to *f(\\a, \\b, \\c)*.
- *For2While.* It replaces *for* loops with *while* loops in Python. To achieve this, we can traverse the AST to identify the initialization, the condition, and the afterthought of *for* loop statement and then add the initialization statement before the while loop.
- *AugAssign.* It operates to unfold some binary assignments e.g., +=, -=, *=, to the regular expressions for example the expression *a += 1* can be mutated to *a = a + 1*.
- *AddDeadCode.* It is designed to insert some dead code fragments such as unused statements or repeated statements in the code. To achieve this, we can repeatedly add the assignment statements after the original statements. For example, there is an assignment *...var = 'abc'...*, we can further add the same statement after this assignment i.e., *...var = 'abc', var = 'abc'...*.
- *VarRename.* It aims to rename the function name or variable names in the code. Following Roziere et al. [61], we rename the function names and variable names with all their occurrence with newly generated names such as *F0*, *V1*, *V2*,... in the code for the implementation.

The defined mutation operations are common in practical scenarios. Developers can easily apply the above simple transformations to the code to meet their requirements. We want to explore whether the detector can still correctly detect the modified content. Investigating whether the detection tools can identify manually modified content is crucial.

3.3.5 Evaluation Metrics. To evaluate the performance of our detectors, we used their AUC scores, FPR and FNR as the evaluation metrics.

AUC score. The AUC score of a detector is interpreted as the probability that the model's ability to accurately classify classes on a scale from 0 to 1, where 1 is best and 0.5 is as good as a random choice. For example, an AUC score of 0.5 implies that the model is only as good as the random choice when assigning probabilities to samples. The higher the AUC score of a classifier, the better its

Table 3: Comparison results of thirteen detectors.

Detector	Model	NLCD-Test							CCD-Test									
			Q&A-LLM			Code2Doc-LLM				CONCODE-LLM			Doc2Code-LLM			APPS-LLM		
		Avg.(AUC)	AUC	FPR	FNR	AUC	FPR	FNR	Avg.(AUC)	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR
	GPT-3.5-Turbo		26.94	0.00	99.95	71.82	35.70	30.32		54.90	31.30	59.90	49.97	77.43	22.01	49.94	79.05	19.16
GPT2_Detector [1]	WizardCoder-15B	47.98	5.78	100.00	27.45	77.57	27.10	30.18	51.63	62.11	27.89	52.59	44.33	0.00	100.00	48.26	76.35	22.22
	CodeLlama-34B-Instruct		30.95	0.00	99.41	74.82	30.96	30.30		62.61	29.10	50.39	47.06	53.74	49.27	45.50	96.53	3.36
	GPT-3.5-Turbo		37.19	7.14	91.17	34.13	37.40	70.73		0.31	100.00	3.90	42.92	56.21	53.40	38.31	61.47	50.82
RoBERTa-QA [32]	WizardCoder-15B	34.80	13.85	89.84	34.13	39.89	19.65	76.35	28.58	0.13	100.00	0.00	42.67	99.98	0.01	48.19	70.03	24.31
	CodeLlama-34B-Instruct		56.61	22.55	60.66	27.13	51.02	71.42		0.05	100.00	2.84	38.06	52.51	65.49	46.55	90.42	8.05
	GPT-3.5-Turbo		82.45	12.85	29.55	94.77	5.93	20.15		79.52	5.75	30.38	49.53	29.69	65.62	47.46	9.33	87.21
ArguGPT [53]	WizardCoder-15B	80.36	25.46	73.79	39.71	93.66	6.03	24.10	57.02	76.91	2.34	34.14	47.87	31.71	68.04	44.55	99.90	0.00
	CodeLlama-34B-Instruct		88.69	7.55	24.57	97.12	3.05	11.80		69.29	2.34	40.38	51.31	27.58	63.67	46.77	17.69	77.68
	GPT-3.5-Turbo		73.99	25.21	36.23	17.83	93.34	22.94		45.31	80.41	11.43	44.72	99.90	0.08	58.22	52.04	34.00
RADAR [38]	WizardCoder-15B	44.81	42.80	43.69	46.52	21.84	91.00	28.19	49.46	33.69	84.24	9.23	50.03	73.30	20.51	67.93	47.20	23.60
	CodeLlama-34B-Instruct		81.55	27.26	18.39	30.86	95.04	3.20		29.21	95.46	3.19	45.33	85.90	12.00	70.70	36.39	30.07
	GPT-3.5-Turbo		74.53	7.32	45.56	89.70	10.73	21.55		60.50	41.23	40.45	50.02	71.55	27.47	49.80	47.04	51.02
MAGE [45]	WizardCoder-15B	78.68	33.21	26.12	77.58	92.11	9.55	18.69	55.64	62.52	25.41	52.24	50.36	70.87	27.91	58.42	26.30	60.91
	CodeLlama-34B-Instruct		91.38	6.36	15.69	91.15	10.63	18.71		62.30	22.07	54.93	50.06	67.17	31.49	56.79	35.83	50.51
	GPT-3.5-Turbo		19.37	95.15	26.26	18.02	97.82	14.78		1.06	0.00	99.93	39.39	99.99	0.01	48.55	67.99	30.12
MultiScale [65]	WizardCoder-15B	24.13	24.89	63.63	58.10	29.63	41.24	74.70	36.51	0.30	100.00	0.28	50.54	55.96	35.41	76.25	13.91	36.85
	CodeLlama-34B-Instruct		29.75	63.17	57.64	23.13	64.94	72.72		0.29	0.00	99.93	39.07	97.89	1.79	73.10	27.57	32.87
	GPT-3.5-Turbo		92.22	12.72	17.70	97.89	5.92	8.63		54.36	68.13	15.40	57.65	57.07	30.49	59.91	63.30	19.62
Fast-DetectGPT [24]	WizardCoder-15B	90.02	55.90	3.84	76.44	96.73	6.91	11.51	72.88	92.21	13.20	17.32	**83.03**	27.22	24.03	71.01	44.85	24.16
	CodeLlama-34B-Instruct		**98.61**	3.48	5.44	98.78	2.00	5.44		81.65	20.72	31.30	78.84	30.45	28.11	**77.26**	36.09	22.73
	GPT-3.5-Turbo		76.00	37.88	9.19	65.43	57.21	2.11		51.14	97.02	0.64	50.79	95.24	3.21	50.90	95.06	3.16
Crossplag.com [14]	WizardCoder-15B	71.42	66.49	39.43	25.11	66.39	57.23	3.83	51.99	50.22	97.02	2.48	53.70	87.54	4.97	52.01	90.47	5.40
	CodeLlama-34B-Instruct		85.92	29.46	4.03	68.26	57.21	0.48		51.27	96.95	0.50	54.76	87.54	2.95	53.12	90.47	3.36
	GPT-3.5-Turbo		73.74	21.09	31.44	93.79	5.88	6.54		31.37	48.83	88.43	50.33	7.62	91.72	49.97	3.11	96.94
GPTZero.me [12]	WizardCoder-15B	76.41	35.09	38.11	91.72	86.13	5.89	21.86	42.84	25.62	48.83	99.93	46.81	6.51	99.87	51.20	2.96	94.65
	CodeLlama-34B-Instruct		76.83	21.09	25.25	92.88	5.88	8.37		26.44	48.83	98.30	50.72	6.51	92.06	53.11	3.01	90.77
	GPT-3.5-Turbo		49.77	100.00	0.23	50.00	100.00	0.00		50.00	100.00	0.00	50.00	100.00	0.00	50.00	100.00	0.00
Grover [13]	WizardCoder-15B	46.83	31.21	100.00	21.00	49.98	100.00	0.02	49.99	50.00	100.00	0.00	50.00	100.00	0.00	50.00	100.00	0.00
	CodeLlama-34B-Instruct		50.00	100.00	0.00	50.00	100.00	0.00		50.00	100.00	0.00	50.00	100.00	0.00	49.95	100.00	0.05
	GPT-3.5-Turbo		94.31	13.59	8.46	87.21	20.16	10.91		45.45	3.48	86.59	57.76	60.32	28.78	58.09	40.88	46.38
Originality.ai [15]	WizardCoder-15B	86.15	57.01	37.33	25.57	88.42	19.65	8.57	54.42	26.08	3.48	94.61	57.39	78.42	7.41	59.31	68.25	14.58
	CodeLlama-34B-Instruct		98.55	3.89	5.12	91.40	13.02	5.57		48.11	3.48	90.84	66.88	51.66	23.69	70.74	43.07	24.46
	GPT-3.5-Turbo		88.67	10.21	21.71	98.66	1.80	3.35		**99.42**	3.26	3.69	60.19	53.05	29.85	26.97	80.53	1.89
Scribbr.com [11]	WizardCoder-15B	88.10	46.42	41.99	43.46	99.06	1.68	1.48	71.20	98.67	1.77	1.99	64.69	39.62	38.44	58.10	38.99	45.01
	CodeLlama-34B-Instruct		96.72	4.21	4.57	**99.07**	1.63	1.25		99.13	1.14	1.42	67.99	39.61	34.48	65.68	41.18	31.09
	GPT-3.5-Turbo		78.08	28.55	28.33	62.10	4.86	58.12		39.56	3.76	90.35	46.93	0.90	98.97	48.39	16.00	82.87
Writer.com [19]	WizardCoder-15B	28.97	0.00	24.06	100.00	6.12	75.66	82.72	22.94	0.00	4.80	100.00	6.02	100.00	0.00	0.00	5.20	100.00
	CodeLlama-34B-Instruct		0.12	93.60	99.18	27.40	43.08	67.85		11.27	64.73	83.25	19.78	78.38	57.45	34.52	55.66	54.38

ability to distinguish between positive and negative classes. We refer to the data generated by LLM as the positive class and the human-generated content as the negative class.

FPR. It refers to the false positive rate, calculated as $FPR = \frac{FP}{FP+TN}$ where FP is the number of false positives (i.e., samples incorrectly classified as LLM-generated), TN is the number of true negatives (i.e., samples correctly classified as human experts) and $N = FP+TN$ is the total number of ground truth negatives (i.e., samples labeled as human experts).

FNR. It refers to the false negative rate, calculated as $FNR = \frac{FP}{FN+TP}$ where FN is the number of false negatives (i.e., samples incorrectly classified as human experts), TP is the number of true positives (i.e., samples correctly classified as LLM-generated) and $P = FN + TP$ is the total number of ground truth positives (i.e., samples are labeled as LLM-generated).

A lower FPR indicates that the model is less likely to mislabel human contents as machine generated, while a lower FNR indicates the reverse. Hence, lower FPR and FNR indicate better performance.

3.3.6 Experiments configuration. All experiments related to GPT were conducted using the OpenAI official API, i.e., the GPT-3.5-turbo-0125 version. For experiments involving open-source models such as WizardCoder-15B and CodeLlama-34B-Instruct, the open-source framework vLLM [43] was utilized. These models were deployed on an NVIDIA DGX system equipped with eight A100 GPUs. A temperature of 0.2 was applied for the generation of data.

4 STUDY RESULTS

In this section, we present the experimental results along with our analysis in an attempt to answer each research question.

4.1 RQ1: Effectiveness of Existing Detectors

The performance of the thirteen detectors on different datasets is shown in Table 3 where the column Avg.(AUC) denotes the average AUC across three different models, and the row Avg.(model-wise) denotes the average value on a specific LLM across different detectors.

In general, the results indicate that detecting LLM-generated content is challenging. On *NLCD-Test*, the average AUC, FPR, and FNR are 61.44, 36.58, and 31.82, respectively. On *CCD-Test*, these values are 49.62, 53.08, and 36.74, respectively. Additionally, the AUC on *NLCD-Test* is higher than on *CCD-Test*, while FPR and FNR are lower. This suggests that detecting LLM-generated code is even more difficult than detecting natural language content, potentially because most existing detectors are trained with more natural language data than code data.

We calculate the average AUC, FPR, and FNR for open-source detectors in the upper part of Table 3 and commercial detectors separately. For open-source detectors, the average values are 53.05, 45.26, and 36.20, respectively, while for commercial detectors, these values are 55.86, 47.89, and 33.10, respectively. These results confirm

Table 4: The AUC performance of different detectors on the Doc2Code-LLM testset in terms of code complexity.

Complexity	Model	Avg.	GPT2_Detector	RoBETa-QA	ArguGPT	MAGE	RADAR	MultiScale	Fast-DetectGPT	GPTZero.me	Writer.com	Scribbr.com	Crossplag.com	Originality.ai
Easy	GPT-3.5-Turbo	50.82	50.36	48.58	45.69	51.36	44.01	40.55	57.91	50.77	47.39	**64.52**	50.26	58.40
	WizardCoder-15B	50.74	50.27	48.06	45.77	51.33	43.75	40.20	58.21	50.68	47.42	**64.63**	50.27	58.30
	CodeLlama-34B-Instruct	50.74	50.37	48.00	46.12	51.55	43.63	40.30	58.13	50.60	47.19	**64.62**	50.22	58.10
Medium	GPT-3.5-Turbo	49.12	49.83	33.34	54.19	49.01	48.76	40.14	56.02	49.66	45.37	**56.35**	50.40	56.35
	WizardCoder-15B	49.13	49.60	33.78	54.11	48.72	48.97	39.22	56.09	49.76	45.54	**56.66**	50.59	56.48
	CodeLlama-34B-Instruct	49.15	49.66	34.33	53.66	48.42	48.11	38.21	56.77	49.87	45.83	**57.15**	50.77	57.04
Hard	GPT-3.5-Turbo	48.45	48.77	35.42	53.63	47.37	43.19	35.06	**59.00**	49.65	47.44	50.34	52.73	58.76
	WizardCoder-15B	48.70	48.94	35.67	53.35	47.61	44.67	36.51	58.19	49.88	47.38	51.36	52.31	**58.53**
	CodeLlama-34B-Instruct	48.80	48.94	35.74	53.27	47.14	44.39	36.62	58.68	49.93	47.71	50.85	52.71	**59.57**
Avg.			49.64	39.22	51.09	49.17	45.50	38.53	57.67	50.09	46.81	57.39	51.14	57.95

Table 5: The AUC performance of different detectors on the Doc2Code-LLM testset in terms of code length.

Length	Model	Avg.	GPT2_Detector	RoBETa-QA	ArguGPT	MAGE	RADAR	MultiScale	Fast-DetectGPT	GPTZero.me	Writer.com	Scribbr.com	Crossplag.com	Originality.ai
Short[0-72]]	GPT-3.5-Turbo	33.69	33.46	36.26	33.41	34.12	36.09	**36.55**	31.40	33.33	32.92	32.33	32.90	31.56
	WizardCoder-15B	50.89	50.87	50.08	49.49	52.90	48.42	46.70	53.50	49.84	45.21	**60.04**	49.81	53.82
	CodeLlama-34B-Instruct	50.68	51.08	49.36	49.91	52.87	47.94	46.17	53.56	49.78	44.64	**59.20**	49.81	53.85
Medium[72-197]	GPT-3.5-Turbo	32.87	33.39	32.08	32.81	33.02	31.80	32.16	33.13	33.32	32.88	**33.49**	33.09	33.25
	WizardCoder-15B	49.66	49.87	41.77	49.14	49.64	42.82	37.64	57.87	50.40	46.87	**60.75**	50.44	58.69
	CodeLlama-34B-Instruct	49.60	49.81	41.47	48.69	49.40	43.06	37.86	57.39	50.44	46.70	**61.04**	50.41	58.89
Long[197+]	GPT-3.5-Turbo	33.44	33.15	31.66	33.78	32.86	32.11	31.29	**35.47**	33.36	34.21	34.18	34.01	35.19
	WizardCoder-15B	50.39	49.03	37.54	49.40	47.73	41.59	33.90	63.23	50.85	50.15	62.80	52.82	**65.65**
	CodeLlama-34B-Instruct	49.96	49.15	38.40	49.69	48.41	41.97	34.59	61.48	50.51	49.02	61.17	52.14	**62.94**
Avg.			44.42	39.85	44.04	44.55	40.64	37.43	49.67	44.65	42.51	51.67	45.05	50.43

Table 6: The AUC performance of different detectors on the Doc2Code-LLM testset on 6 programming languages.

Language	Avg.	GPT2_Detector	RoBETa-QA	ArguGPT	MAGE	RADAR	MultiScale	Fast-DetectGPT	GPTZero.me	Writer.com	Scribbr.com	Crossplag.com	Originality.ai
Go	47.95	46.60	17.50	39.93	48.86	58.61	48.76	**59.82**	50.06	48.21	53.24	52.32	51.51
Java	51.75	50.31	33.78	59.94	52.37	47.12	36.36	63.55	51.82	47.96	**67.58**	50.42	59.80
Javascript	51.56	48.56	53.30	51.58	45.73	40.47	27.23	55.17	49.61	47.67	**80.91**	51.21	67.29
PHP	50.74	49.36	53.29	38.00	50.75	40.48	38.21	59.58	50.79	49.80	**68.13**	50.55	59.99
Python	47.88	52.83	45.54	**58.17**	49.52	41.79	39.91	57.38	48.17	41.70	37.03	50.76	51.80
Ruby	49.98	49.80	**87.28**	45.23	48.04	27.59	33.14	50.06	50.47	40.45	61.95	44.88	60.87
Avg.		49.58	48.45	48.81	49.21	42.68	37.27	57.59	50.15	45.96	61.47	50.02	58.54

that commercial detectors perform better than open-source detectors. Furthermore, within each category, we observe variations in performance. For open-source detectors, Fast-DetectGPT performs the best, while for commercial detectors, Scribbr.com performs better than others.

Further analysis of these detectors reveals that some are ineffective. For example, Grover's FPR value is 100 on the test set, indicating that it classifies all human content as machine-generated content. Similarly, GPTZero.me's FNR value is nearly 100 on the CCD test, meaning that GPTZero.me classifies all machine-generated content as human content, resulting in blind judgments. This comprehensive evaluation across different detectors on different content ensures a thorough understanding of detection performance.

Answers to RQ1: Existing AIGC detectors generally exhibit better performance on natural language data compared to code data, indicating that detecting LLM-generated code is a more challenging task. Despite the superiority of commercial detectors over open-source ones, both still encounter difficulties in accurately detecting LLM-generated code.

4.2 RQ2: Ablation Study of Different Factors

We first investigate the effect of code complexity on detection performance across different detectors. Specifically, we divide the code into easy, medium, and hard categories for testing. The experimental results are presented in Table 4. We observe that as the complexity of the code increases, the AUC also decreases correspondingly, although the extent of the decrease is not very large. Hence, we can conclude that hard code with more complex logic is more challenging for detectors to identify. Furthermore, comparing the detection performance between open-source and commercial detectors across the easy, medium, and hard categories, we find that the average AUC for open-source detectors in terms of easy, medium, and hard codes is 48.29, 47.19, and 46.29, respectively, while for commercial tools, these values are 54.22, 51.85, and 51.94. We can observe that commercial tools have better detection performance on codes of different difficulty levels.

We further investigate the effect of code length on detection performance, and the experimental results are presented in Table 5. We find an interesting phenomenon: for commercial detectors, as the length of the LLM-generated content increases, the detector finds it easier to make the correct judgment. For example, for Originality.ai on the test set of GPT-3.5-Turbo, when the content length increases from short to medium, the AUC increases from 31.56 to 33.25, and finally reaches 35.19 when the content length is long. Similar phenomena also exist in other commercial detectors across different LLM-generated content. However, for open-source detectors, this phenomenon is not as obvious. Intuitively, the longer the model output, the easier it is for the detector to identify that the result is generated by the model. Commercial detectors follow this intuition, while open-source detectors do not. This demonstrates that commercial tools are more reliable than open-source tools.

Finally, we investigate the detection performance of the detectors across different programming languages. The experimental results are presented in Table 6. We observe that some detectors are stable in detecting code written in different programming languages. For

Table 7: Results of fine-tuned models on different *NLCD-Train* datasets.

Detector		NLCD-Test					
		Q&A-LLM			Code2Doc-LLM		
		AUC	FPR	FNR	AUC	FPR	FNR
Unfined-tuned RoBERTa-QA		0.37	0.07	0.91	0.34	0.37	0.71
RoBERTa-QA	Q&A-LLM	1.00	0. 00	0.00	0.69	0.99	0.00
	Code2Doc-LLM	0.84	0.08	0.41	1.00	0.00	0.00
	Composite-NL	1.00	0.00	0.00	1.00	0.00	0.00
	Avg.	0.95	0.03	0.14	0.90	0.33	0.00
MLP	Q&A-LLM	0.89	0.21	0.18	0.42	0.34	0.74
	Code2Doc-LLM	0.43	0.52	0.65	1.00	0.00	0.01
	Composite-NL	1.00	0.01	0.01	0.78	0.34	0.24
	Avg.	0.77	0.25	0.28	0.73	0.23	0.33

Table 8: Results of fine-tuned models on different *CCD-Train* datasets.

Detector		CCD-Test								
		CONCODE-LLM			Doc2Code-LLM			APPS-LLM		
		AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR
Unfined-tuned RoBERTa-QA		0.03	1.00	0.04	0.43	0.56	0.53	0.38	0.62	0.51
RoBERTa-QA	APPS-LLM	0.50	0.00	1.00	0.61	0.39	0.43	0.94	0.49	0.00
	CONCODE-LLM	1.00	0.00	0.00	0.53	0.99	0.00	0.52	1.00	0.00
	Doc2Code-LLM	0.94	0.00	0.94	1.00	0.04	0.01	0.56	0.80	0.06
	Composite-Code	1.00	0.00	0.00	1.00	0.04	0.01	0.84	0.53	0.01
	Avg.	0.86	0.00	0.49	0.78	0.37	0.11	0.71	0.70	0.02
MLP	APPS-LLM	0.29	0.00	1.00	0.51	0.13	0.83	0.73	0.38	0.29
	CONCODE-LLM	0.99	0.02	0.08	0.44	0.98	0.02	0.43	0.68	0.40
	Doc2Code-LLM	0.66	0.48	0.34	0.89	0.22	0.17	0.56	0.45	0.47
	Composite-Code	0.98	0.06	0.13	0.89	0.24	0.16	0.68	0.41	0.32
	Avg.	0.73	0.14	0.38	0.68	0.40	0.30	0.60	0.48	0.37

example, except for Scribbr.com, the remaining commercial detectors demonstrate stability across different programming languages. Some open-source detection tools also exhibit consistent results, such as RoBERTa-QA. However, we notice that the AUC value for Ruby/Go is extremely high/low compared to other programming languages in some cases. Similarly, the AUC value for JavaScript is lower than that of others for MultiScale. We speculate that this imbalance in the data used for training these detectors may contribute to these variations in performance across different programming languages.

Answers to RQ2: An ablation study from a different perspective confirms that commercial tools are indeed more reliable than open-source tools in detection. Generally, the shorter and less complex the generated code, the more challenging it is for detection tools to identify.

4.3 RQ3: Performance of Fine-tuning

Considering that existing detectors primarily focus on natural language content, we further investigate whether fine-tuning code datasets can enhance detection performance for code data. We fine-tuned RoBERTa-QA with different training samples. Since RoBERTa-QA's detection performance in RQ1 is inferior to others, we aim to explore to what extent and how fine-tuning can improve RoBERTa-QA's detection performance in this experiment.

Table 7 presents the results of fine-tuning RoBERTa-QA using various training datasets from *NLCD-Train*. We then evaluate this model on different test sets from *NLCD-Test*. The second row displays the performance of the original detector in Table 3 as a reference for comparison with the three newly fine-tuned detectors. The "Composite-NL" row refers to combining both training sets (i.e., Q&A-LLM and Code2Doc-LLM from *NLCD-Train*). Similarly, Table 8 presents the results of fine-tuning using training datasets from *CCD-Train*.

The results show that fine-tuning can significantly enhance the performance of existing detectors, both for natural language and code in the domain data, i.e., fine-tuning on a training set and then testing on the corresponding test set. It indicates that some underlying patterns still exist in the ChatGPT-generated content, although we have applied some rules for filtering (See Section 3.1). We try to figure out what are the patterns by manually investigating the generated content by ChatGPT, however, we find that it is challenging and we cannot summarize it. We infer that these features might exist in a high-dimensional space and cannot be easily summarized in a low-dimensional space. To confirm this finding, we further design a supplementary experiment to determine whether these patterns can be identified through a simple classifier rather than a deep neural network such as RoBERTa. Specifically, We train a single-layer fully connected neural network (i.e., MLP) as the classifier on the corresponding training set and further test it across different test sets. The experimental results are presented in the last row of Table 7 and Table 8. We can observe that the average AUC score of MLP on NLCD-Test and CCD-Test is 0.75 and 0.67, respectively, which is lower than RoBERTa-QA 0.93 and 0.78. By comparing the performance of MLP and RoBERTa, we can find that shallow neural networks with fewer parameters are worse than deep neural networks, which indicates that the hidden patterns may exist in a high-dimensional space, which is difficult for shallow neural networks to learn.

Furthermore, we observed that fine-tuned models trained on one dataset demonstrate certain generalization capabilities to other datasets. For instance, a model fine-tuned on Q&A-LLM performs better on Code2Doc-LLM compared to its original detector, with the AUC improving from 0.34 to 0.69. Similarly, models trained with CCD-Train show similar improvements. This suggests that fine-tuned models can learn common patterns across different scenarios in LLM-generated content. However, with the increasing value of AUC, FPR also increased accordingly. For example, a model fine-tuned on Q&A-LLM has an FPR (0.99), higher than the unfine-tuned model (0.37) on Code2Doc-LLM. Similar issues also exist on CCD-Test. Thus, we can conclude the generalization capabilities are still limited.

Answers to RQ3: Fine-tuning on the collected ChatGPT-generated content can significantly improve the detection performance of the detectors. We infer that there are some hidden patterns in the high-dimensional space to indicate the content is generated by ChatGPT, however, these patterns are not easily discernible by the naked eye.

4.4 RQ4: Robustness Analysis

In this section, we select the RoBERTa-QA detector that is fine-tuned on the Composite-Code to evaluate the robustness. The results are presented in Table 9, where the row Composite-Code defines the number of samples that the detector can correctly identify in the APPS-LLM test set of GPT-3.5 Turbo. We can observe that the detector can identify 3729 GPT-3.5-generated codes and 1442 Human codes in the APPS-LLM test set. Furthermore, C_{rate} refers to the proportion of the 3729 originally detectable code generated by GPT-3.5 that cannot be detected as the GPT-3.5 generated content after mutation, and H_{rate} refers to the proportion of 1442 originally detectable human-write code that cannot be detected as the human-write content after mutation. The Hybrid refers to randomly selecting one mutation operation from the defined mutation set to modify the code and repeat this process five times.

From Table 9, we can find that in the GPT-3.5-generated code data, the defined mutation operations are able to effectively evade detection by the detector i.e., the detector cannot correctly detect that this is the content generated by GPT-3.5 after the mutation. For example, there are 2096 samples that the detector cannot correctly detect that this content is generated by GPT-3.5 through FuncAddLine mutation to modify the original code. In the defined five mutation operations, we find that VarRename is relatively less effective than others in evading detection in GPT-3.5-generated data. There are only 636 GPT-3.5-generated codes that the detector fails to detect as the GPT-3.5-generated content after the VarRename mutation. We infer that the detector's recognition of whether the content was generated by GPT-3.5 may not be based on variable names. Thus, modifying the variable names has less effectiveness in changing the detector prediction results. Furthermore, We also find that randomly mixing these mutation operators can change the detector's predictions more than using a single operator alone. There are 2704 GPT-3.5-generated codes that the detector fails to predict as the GPT-3.5-generated content. It indicates that mixing multiple mutations are more likely to successfully evade detection.

When referring to the detection performance of the mutation of the human-write content, in the right column of Table 9, we can observe that the defined mutation operators have almost no effect on changing the detector prediction results. For example, there is only one human-write code to be detected as the ChatGPT-generated content by the mutation of AugAssign and For2While. By comparing the mutation performance on the ChatGPT-generated content, we infer that the ChatGPT-generated content is more sensitive than human-write content and it is more susceptible to the influence of mutations.

Answers to RQ4: The mutation operators can effectively change the detection results of content generated by ChatGPT, but they may fail to change the results of human write content. This might be caused by the content generated by ChatGPT is more sensitive.

5 DISCUSSION

5.1 Threats to Validity

Internal Validity. First, the prompts we used to generate the LLM-based content may affect our results. We designed our prompts to mimic what an average user may provide to LLMs under the corresponding usage scenarios. These may not always be the most representative ones. We plan to investigate the effects of different prompts in the future. Furthermore, the answers generated by LLMs are non-deterministic. Different answers may be generated even for the same prompt. But for the purpose of constructing datasets to evaluate AIGC detectors, the impact of non-determinism is limited.

Table 9: Robustness analysis of the fine-tuned models.

Detector	GPT-3.5-Turbo		Human	
	Num	C_{rate}	Num	H_{rate}
Composite-Code	3729	-	1442	-
FuncAddLine	2096	0.56	101	0.07
AugAssign	1219	0.33	1	0.00
For2While	693	0.19	1	0.00
AddDeadCode	1926	0.52	7	0.00
VarRename	636	0.17	54	0.04
Hybrid	2704	0.73	173	0.12

Second, we could not always verify the source of human-provided data in our dataset. For instance, we considered the answers in the Stack Overflow dataset to be provided by human users, but this may not always be true. Some answers could be generated by other tools. Similarly, treating the code snippets in APPS, CONCODE, and Code2Doc as human-written may not always be reliable. But since these datasets were mostly populated with data generated before LLMs became mainstream, we estimate the impact to be limited. Similarly, it is not always possible to verify whether the data from an LLM output is genuinely generated by the model or simply replicates training data from humans. This distinction is particularly challenging when we do not have access to the training data. For the purposes of this paper, all content from LLM outputs is treated as AI-generated data.

Finally, the detectors we studied require detection thresholds to be set, and any threshold chosen may not always work the best in different settings. We acknowledge this potential threat and used the recommended values for each detector to mitigate this issue. Lastly, the filter algorithm may not perform "perfectly", there are still some implicit patterns that exist in LLMs-generated even after filtering, we have found it challenging to definitively pinpoint patterns belonging to AI-generated code. One observed potential pattern that could skip the filter algorithm, which was contributed by our collaborator taking 50 human hours, is that LLMs often format assignment statements with spaces (e.g., a = b), whereas human-written code may not consistently use such spacing (e.g., a=b). Yet, this is not a definitive marker, as formatting can also vary among human coders. We acknowledge some patterns that are obscure to humans, such as lead finetune, which easily reaches saturation.

External Validity. The results obtained on the datasets we used and the detectors we studied may not be generalizable to other data and tools. To mitigate this threat, we collected data from different software development scenarios and contexts to make it more representative of real-world development practice. We selected both commercial and open-source detectors, which we believe represent the state-of-the-art. Lastly, with the rapid development of LLMs, detecting AI-generated content is becoming increasingly difficult. Continuous efforts are needed to improve detection methods. Even if technical detection becomes impossible, governance policies should mandate identification in cases where misuse is a concern.

5.2 Implication

The implications of this study vary across stakeholders. Software developers may increasingly rely on AI-assisted code generation to replace buggy code, with a focus on ethical integration into their workflows. Managers concerned with code quality will face challenges in ensuring AI-generated code meets security, performance, and compliance standards. In education, academic institutions must balance the use of AI tools with concerns over originality, intellectual property, and maintaining academic integrity. Policymakers and regulatory bodies may push for clear guidelines on AI-generated content, including issues related to copyright and accountability. For LLM developers, watermarking techniques could be explored to trace AI outputs, while researchers working on AI content detection may pursue new directions, such as refining detection algorithms and establishing benchmarks for validation [35, 68].

6 RELATED WORK

6.1 AI in Software Engineering

In the early stages of this field, some smaller neural networks with fewer parameters were used to solve the problems in software engineering such as source code summarization [21, 50], vulnerability detection [48, 73] and code search [30, 51]. Different neural network architectures are used, such as LSTMs [37], Transformer [67], and Graph Neural Networks [47]. With the development of this field [35, 71, 72], some more advanced techniques are adopted to achieve higher performance, such as pre-training. The early works for code pre-trained models such as CodeBERT [28], GraphCodeBERT [33] take encoder-only Transformer as the architecture to pre-train a general model on the code-related data and then fine-tune this pre-trained model to downstream tasks to achieve superior performance. The subsequent work has made further improvements such as PLBART [22] and CodeT5 [70], which use encoder-decoder Transformer as the model architecture to improve the model capacity.

Although these pre-trained models have shown significant improvements in different software engineering tasks compared with previous works, they still cannot be applied in a real scenario. OpenAI took a further step and released CodeX model [26], which is trained from the GPT model on publicly available code from GitHub. Furthermore, a distinct production version of CodeX powers GitHub Copilot. ChatGPT released by OpenAI is another representative code, which is fine-tuned from GPT-3.5 series with RLHF for the alignment. The massive knowledge in the GPT-3.5 series and the powerful conversation ability enable ChatGPT to generate accurate answers in different domains. Hence, it can also be applied to software engineering, such as code generation [57], code refinement [34], program repair [42, 63, 64] and program specification generation [55]. For example, Sobania et al. [63] utilized ChatGPT to fix bugs on the standard bug-fixing benchmark QuixBugs [49] and outperformed the state of the art, managing to fix 31 out of 40 bugs. It is precise because ChatGPT has been widely used in software engineering, in this work, we explore whether the ChatGPT-generated code can be detected by existing detectors.

6.2 DeepFake Detection

DeepFake refers to the creation or manipulation of facial appearance through deep generative approaches and deepfake detection aims to identify whether an image or video is synthesized with AI or produced naturally with a camera, which is similar to an AIGC content detector. Based on the extracted features, they can be mainly categorized into spatial-based, frequency-based, and biological signal-based. Detecting deepfake on the spatial domain is the most popular technique in the existing studies [44, 69], and it aims to observe various visible or invisible artifacts on the spatial domain for distinguishing real and fake. Apart from the spatial domain, because the differences between real and synthesized fake faces can also be revealed in the frequency domain, there are also some studies [29, 59] exploiting the differences from the frequency domain. Furthermore, as real facial images and videos produced with cameras are natural compared to synthesized fake faces, biological signals can be used for distinguishing [27, 46, 68]. Compared with deepFake, where the detected objects are images or videos, we aim to identify the text content synthesized by LLMs.

7 CONCLUSION

In this paper, to the best of our knowledge, we are the first to present an empirical study evaluating the performance of existing AIGC detectors in the software domain. We curated a comprehensive dataset of code-related content generated by three state-of-the-art LLMs: GPT-3.5, WizardCoder and CodeLlama. The results of the study indicate that current AIGC detectors struggle with code-related data compared to natural language data. While fine-tuning can improve performance, the generalization of the model still remains a challenge. The findings highlight the need for further research in this area, specifically the development of robust and generalized AIGC detectors.

8 ACKNOWLEDGMENTS

This work is partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23), RIE2025 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) (Award I2301E0026), administered by A*STAR, as well as supported by Alibaba Group and NTU Singapore. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2019. *Openai: GPT-2 Detector.* https://github.com/openai/gpt-2-output-dataset/tree/master/detector

[2] 2020. *Contentatscale: AI DETECTOR.* https://contentatscale.ai/ai-content-detector

[3] 2020. Stackexchange Dataset. (2020). https://github.com/EleutherAI/stackexchange-dataset/blob/master/pairer.py

[4] 2021. *Copyleaks: AI Content Detector.* https://copyleaks.com/ai-content-detector

[5] 2022. *Chatgpt: Optimizing language models for dialogue.* https://chat.openai.com

[6] 2022. *Compilatio: AI Detector Evaluation.* https://ai-detector.compilatio.net

[7] 2022-11. *Stack Exchange (2021-2022): Stack Exchange Data Dump. Archive.org. Dataset.* https://ia800107.us.archive.org/27/items/stackexchange/

[8] 2023. AI Content Detector Accuracy Review. (2023). https://originality.ai/blog/ai-content-detection-accuracy

[9] 2023. *AI Text Classifier.* https://beta.openai.com/ai-textclassifier

[10] 2023. Awesome Chatgpt Prompts. (2023). https://github.com/f/awesome-chatgpt-prompts

[11] 2023. Free AI Detector. (2023). https://www.scribbr.com/ai-detector

[12] 2023. *GPTZero, experiments on july-19 version.* https://gptzero.me

[13] 2023. grover A State-of-the-Art Defense against Neural Fake News. (2023). https://grover.allenai.org/detect

[14] 2023. The most advanced affordable similarity checking tool. (2023). https://crossplag.com/

[15] 2023. Originality AI Plagiarism and Fact Checker. (2023). https://originality.ai/

[16] 2023. *Sapling: AI-Content-Detector.* https://sapling.ai/ai-content-detector

[17] 2023. Unmasking the Wordsmith: How to Tell If a Blog Article Was Written by AI or Human. (2023). https://www.scribbr.com/ai-tools/best-ai-detector/

[18] 2023. *Writefull: GPT Detector.* https://x.writefull.com/gpt-detector

[19] 2023. *Writer: AI Content Detector.* https://writer.com/ai-content-detector

[20] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[22] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[23] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[24] Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. 2023. Fast-DetectGPT: Efficient Zero-Shot Detection of Machine-Generated Text via Conditional Probability Curvature. In *The Twelfth International Conference on Learning Representations.*

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

[26] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[27] Umur Aybars Ciftci, Ilke Demir, and Lijun Yin. 2020. Fakecatcher: Detection of synthetic portrait videos using biological signals. *IEEE transactions on pattern analysis and machine intelligence* (2020).

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[29] Joel Frank, Thorsten Eisenhofer, Lea Schönherr, Asja Fischer, Dorothea Kolossa, and Thorsten Holz. 2020. Leveraging frequency analysis for deep fake image recognition. In *International conference on machine learning.* PMLR, 3247–3258.

[30] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering.* 933–944.

[31] Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How Close is ChatGPT to Human Experts? Comparison Corpus, Evaluation, and Detection. *arXiv preprint arXiv:2301.07597* (2023).

[32] Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arXiv:2301.07597* (2023).

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[34] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering.* 1–13.

[35] Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, Jian Wang, Bing Yu, Wei Feng, and Yang Liu. 2020. Watch out! motion is blurring the vision of your deep neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 975–985.

[36] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[37] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[38] Xiaomeng Hu, Pin-Yu Chen, and Tsung-Yi Ho. 2023. RADAR: Robust AI-Text Detection via Adversarial Learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023.*

[39] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[40] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 2073–2083.

[41] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[42] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *arXiv preprint arXiv:2403.01971* (2024).

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles.*

[44] Haodong Li, Bin Li, Shunquan Tan, and Jiwu Huang. 2020. Identification of deep network generated images using disparities in color components. *Signal Processing* 174 (2020), 107616.

[45] Yafu Li, Qintong Li, Leyang Cui, Wei Bi, Longyue Wang, Linyi Yang, Shuming Shi, and Yue Zhang. 2023. Deepfake text detection in the wild. *arXiv preprint arXiv:2305.13242* (2023).

[46] Yuezun Li and Siwei Lyu. 2018. Exposing deepfake videos by detecting face warping artifacts. *arXiv preprint arXiv:1811.00656* (2018).

[47] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[48] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[49] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity.* 55–56.

[50] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *International Conference on Learning Representations.* https://openreview.net/forum?id=zv-typ1gPxA

[51] Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* (2023).

[52] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[53] Yikang Liu, Ziyin Zhang, Wanyang Zhang, Shisen Yue, Xiaojing Zhao, Xinyuan Cheng, Yiwen Zhang, and Hai Hu. 2023. Argugpt: evaluating, understanding and identifying argumentative essays generated by gpt models. *arXiv preprint arXiv:2304.07666* (2023).

[54] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[55] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. *arXiv preprint arXiv:2401.08807* (2024).

[56] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature. *arXiv preprint arXiv:2301.11305* (2023).

[57] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. 2023. Generating Secure Hardware using ChatGPT Resistant to CWEs. *Cryptology ePrint Archive* (2023).

[58] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[59] Yuyang Qian, Guojun Yin, Lu Sheng, Zixuan Chen, and Jing Shao. 2020. Thinking in frequency: Face forgery detection by mining frequency-aware clues. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XII*. Springer, 86–103.

[60] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[61] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492* (2021).

[62] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.

[63] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[64] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use Chat GPT to Solve Programming Bugs. *International Journal of Information Technology and Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.

[65] Yuchuan Tian, Hanting Chen, Xutao Wang, Zheyuan Bai, Qinghua Zhang, Ruifeng Li, Chao Xu, and Yunhe Wang. 2023. Multiscale Positive-Unlabeled Detection of AI-Generated Texts. arXiv:2305.18149 [cs.CL]

[66] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[68] Run Wang, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yihao Huang, Jian Wang, and Yang Liu. 2019. Fakespotter: A simple yet robust baseline for spotting ai-synthesized fake faces. *arXiv preprint arXiv:1909.06122* (2019).

[69] Sheng-Yu Wang, Oliver Wang, Richard Zhang, Andrew Owens, and Alexei A Efros. 2020. CNN-generated images are surprisingly easy to spot... for now. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8695–8704.

[70] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[71] Xiaofei Xie, Wenbo Guo, Lei Ma, Wei Le, Jian Wang, Lingjun Zhou, Yang Liu, and Xinyu Xing. 2021. RNNRepair: Automatic RNN Repair via Model-based Analysis. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11383–11392. https://proceedings.mlr.press/v139/xie21b.html

[72] Xiaofei Xie, Tianlin Li, Jian Wang, Lei Ma, Qing Guo, Felix Juefei-Xu, and Yang Liu. 2022. NPC: Neuron Path Coverage via Characterizing Decision Logic of Deep Neural Networks. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 47 (apr 2022), 27 pages. https://doi.org/10.1145/3490489

[73] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).