






Attributing ChatGPT-Transformed Synthetic Code

Soohyeon Choi 
University of Central Florida
Orlando, Florida, USA
soohyeon.choi@ucf.edu

Ali Alkinoon 
University of Central Florida
Orlando, Florida, USA
alialkinoon@ucf.edu

Ahod Alghuried 
University of Central Florida
Orlando, Florida, USA
ahodtareq.alghuried@ucf.edu

Abdulaziz Alghamdi 
University of Central Florida
Orlando, Florida, USA
abdulaziz.alghamdi@ucf.edu

David Mohaisen 
University of Central Florida
Orlando, Florida, USA
mohaisen@ucf.edu

Abstract—In this paper, we investigated ChatGPT’s code transformation capability and the effectiveness of the code authorship attribution technique specially designed for ChatGPT code. Through our experiments, we made several key observations. Firstly, ChatGPT demonstrated the capability to transform code in ways that can mislead existing authorship attribution techniques by generating various styles, while it has some constraints, such as the maximum of 12 styles, with certain styles being more commonly employed than others. We also found that the feature-based code authorship attribution proved to be effective when even applied to ChatGPT-transformed code, while the naive approach encountered challenges with accurate classification. In addition, an authorship model trained for binary classification is still effective for ChatGPT-transformed code by achieving up to 93% accuracy. These findings provide insights into the code transformation ability of ChatGPT and shed light on the effectiveness of code authorship attribution techniques for ChatGPT-transformed code.

Index Terms— Code Authorship Identification, Program Stylistic Features, Machine Learning, ChatGPT, Measurement, Code Transformation

I. INTRODUCTION

Large language models (LLMs) are expanding from simple tasks, such as setting reminders or providing basic information (Google Assistant [21], Amazon’s Alexa [4], Apple’s Siri [7], etc.) to sophisticated programming tasks (OpenAI Codex [42], DeepMind AlphaCode [15], etc.) or complex problem-solving (ChatGPT [40], [41], Google Gemini [6], [22], etc.) revolutionizing the way we interact with technology and reshaping various industries. LLMs have significantly advanced their understanding and processing of programming languages [17], [31], [38]. They can now interpret code, identify errors, and provide suggestions for optimization, making them invaluable tools for programmers. Additionally, they can offer tailored recommendations based on a user’s coding style, preferences, and previous projects, enhancing productivity and fostering skill development [33], [55]. Therefore, the use of LLMs as tools for programming language processing, *e.g.*, code generation, code repair, and code completion, has been steadily gaining momentum in recent years [16], [24], [39], [44]. This trend highlights the increasing reliance on LLMs to enhance and streamline various aspects of programming workflows.

ChatGPT, an LLM by OpenAI [40], [41], has been trained on a vast amount of text, including code snippets and examples in various programming languages such as Python, C/C++, and more. The extensive training data allows the model to generate programming code at a level of proficiency comparable to humans [28], [56]. However, these remarkable features also bring potential risks of misuse. For instance, users may exploit ChatGPT to generate solutions for programming assignments, leading to concerns about cheating and plagiarism in academic contexts [5], [25], [35], [36], [57].

Ma *et al.* [36] studied the legal implications of LLMs and the code they generate and highlighted that LLMs rely on both licensed and open-source codes for their training, which could lead to copyright infringement or even negative outcomes resulting from the unregulated utilization of these models. Codes generated by LLMs often exhibit lower security levels or are susceptible to vulnerabilities [37], [46], [50] compared to those generated by humans. Perry *et al.* [46] showed that AI-generated codes often implement lower security features or functions, using basic cipher or short key.

To cope with these issues, codes generated by LLMs need to be identified for further remedies, *e.g.*, to avoid legal and security consequences through proper human vetting, detect plagiarized codes in academic settings, or even attribute malicious codes generated by LLMs. While there has been several studies on code authorship attribution in the context of human-generated codes, employing stylistic and linguistic patterns [1], [2], [11], [12], [26], [30], [32], [53], the problem remains underexplored for LLM-generated code. For instance, among the limited works, Ye *et al.* [59] used code rewriting to identify synthetic (LLM-generated) codes. They conjecture that the rewriting of code generated by LLMs would be minimal in nature. By leveraging the minimal differences between synthetic and rewritten code, compared to the more significant distinctions seen with genuine human-generated code, it may be possible to attribute code to its origin.

Choi *et al.* [12] conducted a study on LLM-generated code produced by ChatGPT and found that ChatGPT can generate code that is highly diverse in terms of style, which presents significant challenges for code authorship attribution.

To address these challenges, they developed a feature-based approach to extract stylistic patterns from ChatGPT-generated code. By grouping the ChatGPT-generated code into sets based on similarities in these extracted features, they trained an authorship attribution model using these sets. This approach successfully attributed authorship to the ChatGPT-generated code with an accuracy of up to 91.7%.

Despite these efforts, these methods remain vulnerable to manipulation through code transformation techniques, which involve modifying the stylistic patterns within code to intentionally deceive the process of identifying its original author [32], [34], [47]. Quiring *et al.* [47] utilized code transformations and leveraged Monte-Carlo Tree Search (MCTS) to evade attribution. Their transformations included control flow, declaration, API transformation, etc., and were applied to the code optimally using the MCTS algorithm. Their code transformations achieved up to 99% evasion attack success rate. Inspired by Quiring *et al.* and Choi *et al.*, our research question centers around the following question:

- **RQ 1:** Is ChatGPT capable of transforming code by incorporating diverse stylistic patterns?

To answer this question, we utilize ChatGPT to transform LLM-generated and human-generated code with two approaches: non-chaining and chaining. Subsequently, we attribute authorship to the ChatGPT-transformed code using a pre-trained code authorship model. Our findings indicate that while there are some limitations, ChatGPT can manipulate stylistic patterns in code effectively, thereby altering the original authorship to that of a different author, contradicting the minimal changes in rewriting conjectured by Ye *et al.* [59]. Since ChatGPT successfully performs such code transformations, it raises another important question:

- **RQ 2:** Can code authorship attribution methods successfully be applied to the ChatGPT-transformed codes?

Due to the diverse styles exhibited by ChatGPT, the conventional code authorship attribution methods are not suitable. As such, we devise a ChatGPT code authorship attribution approach to address the research question. In our initial step, we extracted the stylistic features of ChatGPT-transformed code using a pre-trained code authorship model. Subsequently, we formed sets containing codes that exhibited similar features to evaluate the classification accuracy. Our experiment revealed that, in a standard circumstance, the ChatGPT code authorship approach remained effective for ChatGPT-transformed code, achieving an accuracy of up to 87.5%. Moreover, we developed binary classification models to distinguish between ChatGPT and human-generated code using our dataset. Our model attained an accuracy of up to 93% when applied to the combined dataset and around 91% on three individual datasets.

Contributions. 1) We examined the code transformation of ChatGPT and evaluated the effectiveness of code authorship attribution for ChatGPT-generated code. 2) We introduced two code transformation approaches, non-chaining and chaining, to thoroughly explore ChatGPT’s code transformation capabilities. 3) We provided an extensive analysis of code authorship

attribution for ChatGPT-transformed code and its capacity to achieve a notably high level of classification accuracy. 4) We conducted binary classification with ChatGPT-transformed and non-ChatGPT codes, and the model remains effective on even classifying ChatGPT-transformed codes.

Organization. We review prior work in section II and provide essential background information in section III. Our methodology is detailed in section IV, followed by the experiment setup and goals in section V. Results and discussions are presented in section VI, with the conclusion summarized in section VII.

II. RELATED WORKS

In this section, we provide a detailed overview of the challenges associated with code generated by LLMs. We also delve into the discussion surrounding various code transformation techniques, which are purposefully employed to modify or obscure code in an attempt to mislead or confuse systems designed for code authorship attribution.

A. LLM-generated Code

There has been notable progress in the performance of LLM-powered programming tools in tasks such as synthetic code generation, suggestion, and completion [15], [19], [40], [42], causing various security and ethical issues with LLM use. For example, one ethical concern is that LLMs are trained on a large code base, often “stealing” the coding style of other programmers [36], raising the concern that such use qualifies as plagiarism [5], [25], [35]. More concretely, in an academic context, the use of LLMs to produce programming assignment solutions (or other solutions) is strictly prohibited [29], [57].

LLM-generated code typically demonstrates lower security levels and is more susceptible to vulnerabilities compared with human-generated codes [37], [46]. Perry *et al.* [46] compared LLM-generated codes with human-generated codes to evaluate their security features. They found that the LLM-generated codes rely on basic ciphers, notably substitution ciphers, without performing a fundamental authenticity check on the resulting output [45], [49], [52]. Moreover, LLM-generated codes often employ unsafe randomness when generating codes for specific tasks. In the case of structured query language (SQL) queries, LLMs frequently employed the string concatenation function, which poses a risk of SQL injection attacks [8], [9], [48]. In another study, Botacin [37] demonstrated that LLMs, such as OpenAI’s GPT-3, can be employed to generate a wide variety of functional malware variants that exhibit low detection rates when tested against VirusTotal [13]. While GPT-3 lacks the ability to replicate the complete programs of recent malware attacks, it is nonetheless capable of generating functional versions of older or well-known malware, thereby posing a potential threat [23], [43].

B. Evasion with Code Transformation

Code transformations modify code stylistic features, such as lexical, layout, and syntactic features, with the goal of misattributing its authorship [32], [34], [47].

Quiring *et al.* [47] exploited MCTS to select optimal code transformation to maximize the misattribution. MCTS is a heuristic search determining the best possible moves from diverse options by evaluating the potential value of each individual node in a tree and choosing the next move based on the highest potential value [10], [18]. To use MCTS, the authors introduced multiple code transformations encompassing control flow, declaration, API, template, and miscellaneous transformations. These transformations were sequentially applied to the code to modify its stylistic features. The optimal transformed code is selected using MCTS while adhering to certain constraints, such as minimizing the number of transformations applied or maximizing the score strand deviation. Through this approach, MCTS demonstrated an impressive success rate of up to 99.2% in evading detection during attacks, all while maintaining the original functionality of the code sample.

Li *et al.* [32] concealed code stylistic patterns by extracting coding style attributes such as keyword usage, indentation, and variable naming conventions for each author. These attributes are then used to select a specific style, which is applied to the code using a multi-language parsing tool called srcML [51]. The outcome is a modified code that closely resembles the coding style of the desired author.

III. BACKGROUND

Human-generated code authorship has been addressed in numerous studies, while synthetic (LLM-generated) codes are less so. In the following, we review the background such as code authorship attribution, ChatGPT, and our motivation.

A. Code Authorship Attribution

There has been a significant focus on researching and creating practical tools for accurately attributing code authorship in the academic literature [1]–[3], [26], [27], [32]. For instance, Caliskan-Islam *et al.* [26] used the random forests (RF) techniques along with distinctive linguistic stylistic aspects of code to determine the authorship of code. They started by extracting different types of stylistic features from a given code and used them to obtain stylometry data, which is treated as a distinct attribute associated with each author. The features they extracted include lexical, layout, and syntactic. The lexical and layout features can be derived from the code, whereas syntactic are obtained from an abstract syntax tree (AST).

Lexical features are identified by analyzing naming conventions, such as variable and function names. The length of these names can offer insights into a developer’s coding style—short names may indicate a focus on brevity, while longer names suggest clarity and maintainability. Certain keywords or patterns in names may also reveal the use of specific programming paradigms or design patterns, with prefixes in names reflecting developer preferences.

Layout features involve formatting and stylistic choices made by authors, including aspects such as indentation style and comment usage. For instance, some authors may prefer two spaces for indentation, while others opt for four spaces.

Additionally, their preference for single-line versus multi-line comments, as well as the specific way they format brackets and other structural elements, serves as a distinctive reflection of their individual coding style.

Syntactic features pertain to the syntax of the code, including language-specific properties *e.g.*, the AST structure and the usage patterns of keywords. These features are typically identified through an analysis of the code’s structural elements, such as the maximum depth of the AST, the frequency of specific language constructs, and other syntax-related metrics that provide insights into the code’s composition.

The extracted features are converted into a vector format and subsequently utilized to train an RF classifier designed to identify and distinguish between code authors based on their unique coding characteristics. This approach proved highly effective, achieving an accuracy of over 90% when tested on a dataset comprising code samples from 1,600 distinct authors.

Since then, numerous efforts have been made in this domain to enhance and scale authorship attribution by leveraging a variety of machine learning methods and representation techniques [1]–[3], [27], [32]. In this study, we focus on the work of Caliskan-Islam *et al.*, as it forms the foundation for the experiments conducted in the remainder of this work.

B. ChatGPT

ChatGPT, powered by the GPT-4.0 architecture developed by OpenAI [40], [41], stands out as an exceptional interactive large language model with advanced writing capabilities in both natural and programming language. ChatGPT generates responses closely resembling human responses, making it an invaluable tool in various domains. Through extensive training on an enormous corpus of over 750 gigabytes, ChatGPT has an extensive knowledge base that allows it to provide detailed responses to a wide range of prompts.

A key strength of ChatGPT lies in its exposure to numerous code snippets during training [58]. This exposure has equipped the model with the ability to comprehend and produce human-like responses to programming-related queries, including troubleshooting code, exploring programming concepts, or seeking guidance on software development. To refine and optimize its performance, ChatGPT undergoes training and fine-tuning using the reinforcement learning from human feedback (RLHF) technique [14], [54]. This approach involves an AI trainer who takes on the roles of both the user and the LLM in simulated interactive conversations. Using this methodology, the model undergoes supervised fine-tuning based on detailed, human-guided feedback provided during these interactions. Through RLHF, ChatGPT benefits from an iterative training process, enabling it to progressively refine its capabilities and deliver responses that are more accurate, contextually relevant, and coherent. By leveraging feedback from human trainers, ChatGPT becomes increasingly proficient with responses that align more closely with user expectations.

C. Motivation

The use of LLMs such as ChatGPT as a programming assistant may raise potential ethical and security concerns [5],

[25], [35], [37], [46]. Consequently, it becomes crucial to accurately attribute ChatGPT-generated codes to effectively address these concerns. However, the training of LLMs involves exposure to multiple authors' styles [58], which can result in the generation of code in various styles, thereby making it challenging to attribute authorship to a specific style. Moreover, the ability of LLMs to generate programming codes in diverse styles can lead to code transformations that may deceive or misrepresent the true authorship of the code. This introduces additional challenges in accurately identifying the origin of the code, complicating efforts to address issues such as intellectual property rights, the detection and prevention of plagiarism and the code integrity.

To tackle these challenges, we analyzed ChatGPT's code transformation capability by examining the stylistic features of codes transformed by ChatGPT. We applied a code authorship attribution method specifically tailored for ChatGPT-generated code to assess the effectiveness of the model in accurately attributing authorship to such ChatGPT transformed code samples. By investigating these aspects, we aimed to gain insights into ChatGPT's ability to transform code and its suitability for code authorship attribution.

D. Threat Model

The objective of this work is to assess ChatGPT's capacity to perform code transformations on both codes generated by ChatGPT and written by humans (in the rest of this paper, we will refer to them by non-ChatGPT), while also evaluating its ability to produce different coding styles to address issues of ChatGPT. Therefore, in our experiment, we assume that an adversary can utilize the ChatGPT model and modify the stylistic patterns of code. The adversary's objective is to alter the code using ChatGPT to misattribute its original author. We utilized ChatGPT's API as the publisher provided [40] without any modification. This allows the adversary the full capabilities of ChatGPT, providing them with the necessary tools and capabilities to carry out their intended objective.

IV. OUR METHODOLOGY

We employed ChatGPT [40] as our main LLM for code transformation purposes and the code authorship attribution model developed by Caliskan-Islam *et al.* [26] as our baseline authorship method. Building upon Caliskan-Islam *et al.*'s work, we build an improved approach designed explicitly for ChatGPT code authorship attribution. This approach was developed to address the challenges of applying existing code authorship models to AI-generated codes addressing code diversity, which we employed.

A. ChatGPT Code Authorship

Due to LLMs' ability to generate codes with various styles based on what they are trained on, Choi *et al.* [12] raised a question regarding the effectiveness of off-the-shelf code authorship attribution methods in identifying the authorship of code generated by ChatGPT. This concern comes from the fact that ChatGPT was trained using a vast collection of code

written by multiple authors, thus, it can generate codes in multiple styles, which poses a significant challenge for the current attribution methods.

To address this question we conduct experiments by initially generating source codes in C++ format with ChatGPT by utilizing challenges from Google Code Jam (GCJ) [20], especially in the years 2017, 2018, and 2019. GCJ is an annual programming competition organized by Google. The goal of GCJ is to solve a series of challenging algorithmic problems within a specified time limit and participants compete in multiple online rounds, with each round consisting of several problems. Thus, we obtained challenge statements, constraints, and sample input and output—eight challenges per year—and utilized them to generate codes by ChatGPT. Additionally, we collected code samples from participants, a total of 204 authors per year, that corresponded to each year's challenges and used them as non-ChatGPT codes.

With the ChatGPT-generated code, we analyzed codes' unique stylistic patterns by utilizing an authorship model, which was trained using non-ChatGPT code. This attribution model was trained with code samples from 204 different authors, enabling it to differentiate between 204 distinct styles. Consequently, this model can serve as an oracle to identify and narrow down the stylistic patterns present in the code generated by ChatGPT.

From this experiment, it is observed that ChatGPT can generate source code in various styles from the same question (challenge statement) while there is a limited range of styles. For instance, ChatGPT generated a total of 500 codes from one challenge statement, but only up to 27 different styles were observed. This finding suggests that although ChatGPT is able to generate code in various styles, it would be feasible to identify ChatGPT-generated code's authorship if the model learns common features of them.

1) *Feature-based Approach*: The feature-based approach to ChatGPT code authorship consists of three main stages: analyzing the stylistic features, grouping the codes, and training a new model. Initially, the stylistic features of ChatGPT-generated code are examined using a non-ChatGPT authorship model. Based on the results of this initial analysis, we created sets of ChatGPT-generated code that exhibit similar stylistic features. These sets were then combined with non-ChatGPT code samples. Finally, a new authorship model was trained using the combined dataset, enabling it to learn the distinctive features of ChatGPT-generated code.

2) *Naive Approach*: We carried out experiments using a naive approach, which involves using only the initial response from ChatGPT without considering its stylistic patterns. This approach is based on the assumption that when utilizing ChatGPT, users typically exhibit a natural behavior to accept the first response provided by the model, unless they identify an error or find it to be incorrect.

The observations derived from these experiments indicated that the model employing the feature-based method achieved an impressive classification accuracy of over 93%, demonstrating its effectiveness. In contrast, the model that relied on the

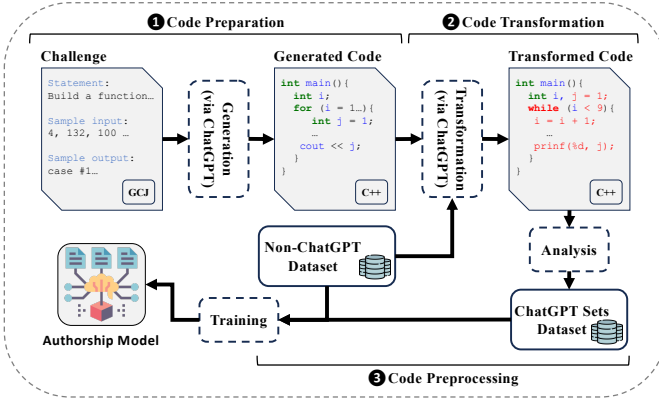


Figure 1: An overview of ChatGPT code transformation.

naive approach struggled significantly with classification tasks, achieving a much lower accuracy of only 29.2%.

B. ChatGPT Code Transformation

Inspired by the code transformation technique and the ChatGPT code authorship attribution approach highlighted earlier, this study focused on a comprehensive examination of ChatGPT’s ability to transform code. It also evaluates the effectiveness of ChatGPT’s authorship attribution method when applied to code that has undergone transformation using ChatGPT, with a focus on how well the model can still identify the original author of the transformed code.

The pipeline of the transformation is shown in Figure 1. We employed the same challenge sets from GCJ 2017, 2018, and 2019 used earlier. With the sets of challenges, we generated one code per each challenge statement and utilized them as “ChatGPT-generated code”. Also, we selected one author from each year of the GCJ participant dataset and used them as “non-ChatGPT codes” (①). We then applied transformations on both the ChatGPT-generated code and the non-ChatGPT code using ChatGPT itself. We presented each piece of code to ChatGPT and requested it to alter its stylistic features, such as variable and function names, code structures, and so on (②). With ChatGPT-transformed code, we analyzed the stylistic features (predicted labels) of them by exploiting pre-trained non-ChatGPT authorship model to create sets of ChatGPT-transformed code that exhibit similar features by utilizing the feature-based approach. We also created sets of ChatGPT-transformed code using the naive approach for comparison. We then combined the sets with the non-ChatGPT dataset, respectively, to train a new authorship model for ChatGPT code authorship for transformed codes (③).

For the code transformation process, we developed and implemented two distinct approaches: non-chaining transformation (NCT) and chaining transformation (CT). These methods were designed to explore different ways of altering code while preserving its functionality, allowing for a comprehensive analysis of their effects on authorship attribution.

Non-chaining Transformation (NCT). NCT applies multiple transformations to the code *repeatedly* as shown in Figure 2 in blue. Given an initial ChatGPT code CG_{C_0} and ChatGPT

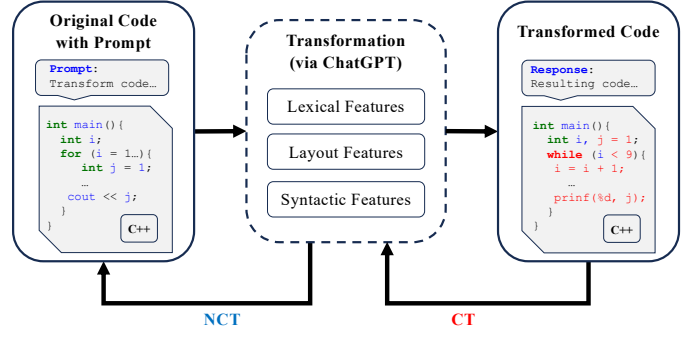


Figure 2: Non-chaining (NCT) vs. chaining transformation (CT).

```
int main() {
    int nCase;
    cin >> nCase;
    for (int iCase = 1; iCase <= nCase; ++iCase) {
        int d, n, double t = 0;
        cin >> d >> n;
        for (int i = 0; i < n; ++i) {
            int x, y;
            cin >> x >> y;
            x = d - x;
            t = max(t, (double)x / (double)y);
        }
        cout << iCase << (double)d / (double)t;
    }
}
```

Figure 3: Example of original code for transformation by ChatGPT.

transformation function GPT , NCT with ChatGPT code can be denoted as follows: $CG_{C_i} = GPT(CG_{C_0}); \forall i(1 \leq i \leq 50)$. Given an initial non-ChatGPT code NCG_{C_0} and ChatGPT transformation function GPT , non-chaining transformation with non-ChatGPT code can be defined as: $NCG_{C_i} = GPT(NCG_{C_0}); \forall i(1 \leq i \leq 50)$.

For instance, when prompted to transform the original code (Figure 3), ChatGPT generates a transformed version of the code (Figure 4a). Then, when provided with a new prompt to modify the same original code (Figure 3), it produces a different transformed code (Figure 4b). This demonstrates the model’s ability to create multiple variations of code based on the same initial prompt.

Chaining Transformation. Chaining transformation involves the sequential process of transforming different codes each time. In this approach, the output of one code transformation becomes the input for the next transformation. This process is visually illustrated in Figure 2, with the transformations highlighted in red to indicate the flow from one to the next.

Given an initial ChatGPT code, denoted as CG_{C_0} , and its transformed code, called CG_{C_1} , along with a transformation function GPT , CG_{C_1} will be an input of the next transformation. We represent the process of applying chaining transformations to the ChatGPT code as: $CG_{C_{i+1}} = GPT(CG_{C_i}); \forall i(0 \leq i \leq 49)$. Moreover, if we have an initial non-ChatGPT code, referred to as NCG_{C_0} , and its transformed code NCG_{C_1} , along with a transformation function GPT . Then NCG_{C_1} will be an input of the next transformation. We can describe the process of applying individual transformations

```

double solveTestCase(int d, int n){
    double maxTime = 0;
    for (int i = 0; i < n; ++i){
        int x, y;
        cin >> x >> y;
        x = d - x;
        maxTime = max(maxTime, (double)x / (double)y);
    }
    return (double)d / maxTime;
}
int main(){
    int numCase;
    cin >> numCase;
    for (int iCase = 1; iCase <= numCase; ++iCase){
        int distance, numHorses;
        cin >> distance >> numHorses;
        double result = solveTestCase(distance,
            NumHorses);
        printf("Case #d: %.6lf\n", iCase, result);
    }
}

```

(a) The first transformed code of the original code in Figure 3 with NCT.

```

double solveTestCase(int caseNumber) {
    int maxDistance, numHorses;
    cin >> maxDistance >> numHorses;
    double maxTime = 0.0;
    for (int i = 0; i < numHorses; ++i) {
        int horsePosition, horseSpeed;
        cin >> horsePosition >> horseSpeed;
        int position = maxDistance - horsePosition;
        double timeToFinish = (double)(position /
            horseSpeed);
        maxTime = max(maxTime, timeToFinish);
    }
    double maxSpeed = (double)maxDistance / maxTime;
    printf("Case #d: %.6lf\n", caseNumber, maxSpeed);
}
int main() {
    int numTestCases;
    cin >> numTestCases;
    for (int i = 1; i <= numTestCases; ++i) {
        solveTestCase(i);
    }
}

```

(b) The second transformed code of the original code in Figure 3 with NCT.

Figure 4: Examples of ChatGPT-transformed codes using Non-Chaining Transformation (NCT).

Table I: Non-ChatGPT code datasets that are used to train non-ChatGPT code authorship models.

| Dataset | Authors | Challenges | Language | Total |
|----------|---------|------------|----------|-------|
| GCJ 2017 | 204 | 8 | C++ | 1,632 |
| GCJ 2018 | 204 | 8 | C++ | 1,632 |
| GCJ 2019 | 204 | 8 | C++ | 1,632 |

to the non-ChatGPT code in the following: $NCGc_{i+1} = GPT(NCGc_i); \forall i_{(0 \leq i \leq 49)}$. When given a prompt to modify the original code (3), CT generates a transformed code (5a). Subsequently, the initial transformed code (5a) is used as an input, and ChatGPT produces a new transformed code (5b).

```

double solveTestCase(int caseNumber) {
    int d, n;
    cin >> d >> n;
    double maxTime = 0;
    for (int i = 0; i < n; ++i) {
        int x, y;
        cin >> x >> y;
        x = d - x;
        maxTime = max(maxTime, (double)x / (double)y);
    }
    return (double)d / maxTime;
}
int main() {
    int nCases;
    cin >> nCases;
    for (int iCase = 1; iCase <= nCases; ++iCase) {
        double result = solveTestCase(iCase);
        printf("Case #d: %.6lf\n", iCase, result);
    }
}

```

(a) The first transformed code of the original code in Figure 3 with CT.

```

double solve_test_case(int case_number) {
    int d, n;
    cin >> d >> n;
    double max_time = 0;
    for (int i = 0; i < n; ++i) {
        int x, y;
        cin >> x >> y;
        x = d - x;
        max_time = max(max_time, (double)x / (double)y);
    }
    return (double)d / max_time;
}
int main() {
    int num_cases;
    cin >> num_cases;
    for (int case_num = 1; case_num <= num_cases; ++
        case_num) {
        double result = solve_test_case(case_num);
        cout << "Case #" << case_num << ": " << result;
    }
}

```

(b) The second transformed code of the first transformed code in Figure 5a with CT.

Figure 5: Examples of ChatGPT-transformed codes using Chaining Transformation (CT).

Table II: ChatGPT code datasets that are transformed by ChatGPT with ChatGPT-generated and non-ChatGPT code using the non-chaining and chaining approach per each challenge (a total of 8 challenges per year). +N stands for ChatGPT +NCT, +C for ChatGPT +CT, \pm N for Non-ChatGPT +NCT, \pm C for Non-ChatGPT +CT.

| Dataset | +N | +C | \pm N | \pm C | Total |
|----------|----|----|---------|---------|---------------|
| GCJ 2017 | 50 | 50 | 50 | 50 | 1,600 (200x8) |
| GCJ 2018 | 50 | 50 | 50 | 50 | 1,600 (200x8) |
| GCJ 2019 | 50 | 50 | 50 | 50 | 1,600 (200x8) |

V. EXPERIMENT SETUP AND GOAL

For code transformation tasks with ChatGPT, we applied the NCT and CT to both the ChatGPT code and non-ChatGPT code. Moreover, we used an authorship attribution method proposed by Caliskan-Islam [26] with the feature-based approach and naive approach mentioned earlier. Finally, we

Table III: List of datasets used for the binary classification (ChatGPT vs. Human) experiments.

| Dataset | # of challenges | # of codes | Language | Total |
|----------|-----------------|------------|----------|-------|
| GCJ 2017 | 8 | 200 | C++ | 3,200 |
| GCJ 2018 | 8 | 200 | C++ | 3,200 |
| GCJ 2019 | 8 | 200 | C++ | 3,200 |
| Combined | 15 | 200 | C++ | 6,000 |

conducted the binary classification which involved only two labels (ChatGPT vs. Human) with the transformed codes by ChatGPT and the GCJ datasets.

A. Experiment Setup

We conducted our experiments on a workstation running Ubuntu 20.04.5 LTS. The hardware setup included an Nvidia RTX A6000 48GB GPU and an Intel Core i7-8700K CPU.

B. Datasets

Non-ChatGPT Dataset. To create the non-ChatGPT dataset, we gathered eight codes representing eight different challenges (problem statements). These codes were written by 204 authors from each year, specifically GCJ 2017, 2018, and 2019. Consequently, our non-ChatGPT dataset for the GCJ 2017, 2018, and 2019 years comprises a total of 1,632 code samples, respectively as indicated in Table I.

ChatGPT Dataset. For the ChatGPT dataset, we began by generating a single code for each challenge (a total of eight challenges). These initial codes were then transformed using ChatGPT, employing both the NCT and CT approaches. Moreover, we selected a single author along with their eight codes from each year of the non-ChatGPT dataset and transformed their codes with ChatGPT as well. This resulted in a compilation of 50 codes for each setting—ChatGPT code with NCT and CT, non-ChatGPT code with NCT and CT—, totaling 200 codes per year. Thus, we compiled a total of 1,600 code samples per year (*i.e.*, 200 code samples for each of the eight challenges) as shown in Table II.

Binary Classification Dataset. For binary classification experiments, we utilized the GCJ datasets from the years 2017, 2018, and 2019, in addition to the code samples transformed using ChatGPT. Each of these datasets consisted of two distinct classes: ChatGPT and human, and each class contained 1,600 code samples. These samples were evenly distributed across eight challenges, comprising 200 code samples per challenge. Consequently, each year’s dataset totaled 3,200 code samples. Moreover, we combined these datasets from the three years into a single dataset and performed an experiment. However, in this combined dataset, we reduced the number of challenges per year from eight to five, resulting in a total of 6,000 code samples to maintain a balanced number of code samples within each dataset. Keeping eight challenges would have resulted in an imbalance compared to the other datasets. You can find detailed information about these datasets in III.

C. Experiments

To conduct the experiment, we initially obtained pre-trained non-ChatGPT authorship attribution models. These models

Table IV: The number of styles refers to the total number of predicted labels that are assigned to ChatGPT-transformed code by pretrained non-ChatGPT authorship model. +N stands for ChatGPT +NCT, +C for ChatGPT +CT, \pm N for Non-ChatGPT +NCT, \pm C for Non-ChatGPT +CT, and A for average.

| | GCJ 2017 | | | | GCJ 2018 | | | | GCJ 2019 | | | |
|----|----------|-----|---------|---------|----------|-----|---------|---------|----------|-----|---------|---------|
| | +N | +C | \pm N | \pm C | +N | +C | \pm N | \pm C | +N | +C | \pm N | \pm C |
| C1 | 4 | 3 | 2 | 1 | 3 | 2 | 6 | 4 | 3 | 3 | 7 | 3 |
| C2 | 1 | 1 | 1 | 1 | 7 | 3 | 12 | 2 | 2 | 1 | 10 | 1 |
| C3 | 5 | 2 | 1 | 1 | 3 | 2 | 11 | 6 | 4 | 2 | 7 | 4 |
| C4 | 4 | 1 | 4 | 1 | 6 | 2 | 12 | 4 | 4 | 1 | 11 | 3 |
| C5 | 3 | 2 | 5 | 4 | 1 | 1 | 7 | 3 | 5 | 2 | 5 | 1 |
| C6 | 2 | 1 | 3 | 4 | 5 | 1 | 11 | 2 | 2 | 1 | 5 | 3 |
| C7 | 3 | 1 | 2 | 1 | 2 | 2 | 7 | 2 | 3 | 1 | 6 | 2 |
| C8 | 3 | 3 | 2 | 3 | 4 | 1 | 11 | 7 | 3 | 1 | 6 | 2 |
| A | 3.1 | 1.8 | 2.5 | 2 | 3.9 | 1.8 | 9.6 | 3.8 | 3.3 | 1.5 | 7.1 | 2.4 |

were trained using non-ChatGPT datasets from GCJ 2017, 2018, and 2019, respectively. Next, we extracted stylistic features from the ChatGPT-transformed codes by exploiting the non-ChatGPT authorship models. This allowed us to create sets of the ChatGPT-transformed codes that exhibited similar features using the feature-based approach. In addition, we made sets of ChatGPT-transformed codes using the naive approach for comparison with the feature-based approach. In the final step, we categorized all ChatGPT-generated codes under a single label, denoted as “ChatGPT” and categorized all non-ChatGPT codes under the label “human”. This categorization was done to facilitate a binary classification experiment, allowing for a clear distinction between machine-generated and human-written code.

For evaluation, we analyzed the ChatGPT-transformed codes’ stylistic patterns by examining the number of styles and the diversity of styles that ChatGPT generated. Additionally, we combined the sets of ChatGPT-transformed codes with the non-ChatGPT code dataset. This combined dataset was then used to train a new authorship model, enabling us to evaluate the effectiveness of the ChatGPT code authorship attribution model for the ChatGPT-transformed code. In the context of binary classification, we assessed classification accuracy, which measures how effectively the trained model correctly identifies the label for each code.

VI. RESULTS AND DISCUSSION

We examined the number of styles that ChatGPT generated to understand their diversity. Then, we examined the styles within ChatGPT-transformed code to gain more comprehensive insights into the range of coding styles generated (transformed) by ChatGPT. Finally, we tested the classification accuracy of the ChatGPT code authorship attribution method using both ChatGPT-transformed and non-ChatGPT code samples.

A. The Number of Styles

The term “number of style” refers to the count of distinct predicted labels assigned to ChatGPT-transformed code samples by the non-ChatGPT model. We first trained models using non-ChatGPT datasets specific to GCJ 2017 and 2018. Subsequently, we used these models to predict labels for the ChatGPT-transformed code samples.

Through these experiments, we made an observation that ChatGPT possesses the capability to transform code. However, it is important to note that there are certain limitations associated with this ability. In the experiment involving the ChatGPT-transformed code of GCJ 2017, the results indicated that the maximum number of styles observed was limited to 5. On average, across different settings, the number of styles per setting was found to be 3.1, 1.8, 2.5, and 2 for ChatGPT with NCT, ChatGPT with CT, Non-ChatGPT with NCT, and Non-ChatGPT with CT, respectively. These findings are summarized in Table IV. With the ChatGPT-transformed code of GCJ 2018, our observation revealed that the maximum number of styles identified was 12. Furthermore, the average number of styles for each setting was found to be 3.9, 1.8, 9.6, and 3.8 for ChatGPT with NCT and CT, and Non-ChatGPT with NCT and CT, respectively as presented in Table IV.

B. Confirmation of The Number of Styles

The analysis revealed that the maximum number of styles and average number of styles for the GCJ 2018 dataset were higher compared to the GCJ 2017. Additionally, it was observed that the setting with the highest average number of styles for the GCJ 2017 was ChatGPT with NCT, whereas, for the GCJ 2018, the highest number of styles was observed in the non-ChatGPT with NCT setting. Based on the obtained results, it is evident that ChatGPT possesses the ability to transform code. However, these findings also revealed a lack of consistency. To further validate and confirm these observations, an additional experiment was conducted using the ChatGPT-transformed code from the GCJ 2019 dataset.

The findings from the GCJ 2019 experiment demonstrated that the maximum number of styles was 11. Additionally, the average number of styles for each setting was 3.3, 1.5, 7.1, and 2.4 for ChatGPT with NCT and CT, and non-ChatGPT with NCT and CT, respectively as presented in Table IV. The patterns observed in the results of the GCJ 2019 experiment were similar to those of the GCJ 2018 experiment. Specifically, the maximum number of styles and the highest average number of styles were both found in the non-ChatGPT with NCT setting. This consistency in patterns further reinforces the observations made in the previous experiments.

C. Diversity of Styles

Based on ChatGPT’s code transformation capability observed from the previous experiments, we examined the diversity of styles that ChatGPT produced. The diversity of styles refers to how frequently each label was utilized, providing insights into the variety of coding styles generated. For the GCJ 2017 experiment, one specific label (Author49) accounted for a substantial portion of 77.1%, indicating that this label appeared 1,234 times out of 1,600 code samples as presented in Table V. In the case of the GCJ 2018 experiment, three labels (Author64, Author135, and Author19) held proportions of 24.8% (397), 23.4% (375), and 18.3% (293), respectively, which collectively amounted to 66.5% of the total. These results can be found in Table VI. With the GCJ 2019 experiment,

Table V: The diversity of styles – GCJ 2017. The result filters all labels with less than two occurrences (which was a total of 8).

| Label | Occurrences | Percentage | Label | Occurrences | Percentage |
|-------|-------------|------------|-------|-------------|------------|
| A49 | 1234 | 77.1 | A67 | 23 | 1.4 |
| A98 | 62 | 3.8 | A202 | 15 | 0.9 |
| A197 | 48 | 3.0 | A115 | 11 | 0.6 |
| A80 | 42 | 2.6 | A18 | 6 | 0.3 |
| A139 | 41 | 2.5 | A57 | 6 | 0.3 |
| A23 | 35 | 2.1 | A92 | 5 | 0.3 |
| A58 | 32 | 2.0 | A157 | 4 | 0.2 |
| A16 | 25 | 1.5 | A198 | 3 | 0.1 |

Table VI: The diversity of styles – GCJ 2018. The result filters all labels with less than two occurrences (which was a total of 19).

| Label | Occurrences | Percentage | Label | Occurrences | Percentage |
|-------|-------------|------------|-------|-------------|------------|
| A64 | 397 | 24.8 | A6 | 12 | 0.7 |
| A135 | 375 | 23.4 | A196 | 8 | 0.5 |
| A19 | 293 | 18.3 | A12 | 7 | 0.4 |
| A1 | 98 | 6.1 | A145 | 6 | 0.3 |
| A121 | 93 | 5.8 | A51 | 6 | 0.3 |
| A166 | 46 | 2.8 | A141 | 6 | 0.3 |
| A84 | 39 | 2.4 | A160 | 6 | 0.3 |
| A154 | 28 | 1.7 | A53 | 5 | 0.3 |
| A18 | 28 | 1.7 | A59 | 5 | 0.3 |
| A128 | 28 | 1.7 | A47 | 5 | 0.3 |
| A133 | 24 | 1.5 | A139 | 5 | 0.3 |
| A120 | 19 | 1.1 | A4 | 5 | 0.3 |
| A183 | 16 | 1.0 | A29 | 3 | 0.1 |
| A114 | 15 | 0.9 | A190 | 3 | 0.1 |

two labels (Author9 and Author31) accounted for 39.9% (639) and 18.7% (300) of the total instances, respectively, summing up to 58.6% of the total. Furthermore, there are following three labels (Author27, Author106, and Author118) appeared 8.3% (134), 8.3% (134), and 8.2% (132), respectively in Table VII.

The patterns observed in the GCJ 2018 and 2019 experiments were similar, suggesting consistency in the distribution of coding styles between these two datasets. This similarity further strengthens the observed patterns across multiple experiments. Additionally, the results of the GCJ 2017 experiment stood out as being different from the others. It was noted that a single label, “Author49”, accounted for over 77% of the total instances. This significant imbalance in label distribution could be a contributing factor to the divergent results observed in the GCJ 2017 at the previous experiment. The dominance of a single label limited the overall diversity of coding styles in that dataset, leading to distinct outcomes compared to the other experiments such as the GCJ 2018 and 2019.

D. Authorship Attribution Accuracy

To facilitate our analysis, we employed the feature-based approach and utilized the stylistic features (predicted labels) extracted from our experiments. This allowed us to create a set of ChatGPT-transformed code samples that shared similar stylistic characteristics for each of the datasets (GCJ 2017, 2018, and 2019). These transformed code samples were then combined with non-ChatGPT datasets which include 204 authors’ code samples, thus a total of 205 (204 non-ChatGPT authors plus one ChatGPT set) specific to each respective dataset (GCJ 2017, 2018, and 2019). Subsequently, we utilized these combined datasets to train new authorship attribution models, aiming to assess their classification accuracy in attributing

Table VII: The diversity of styles – GCJ 2019. The result filters all labels with less than two occurrences (which was a total of 9).

| Label | Occurrences | Percentage | Label | Occurrences | Percentage |
|-------|-------------|------------|-------|-------------|------------|
| A9 | 639 | 39.9 | A33 | 13 | 0.8 |
| A31 | 300 | 18.7 | A95 | 12 | 0.7 |
| A27 | 134 | 8.3 | A86 | 11 | 0.6 |
| A106 | 134 | 8.3 | A183 | 10 | 0.6 |
| A118 | 132 | 8.2 | A54 | 9 | 0.5 |
| A1 | 63 | 3.9 | A152 | 7 | 0.4 |
| A45 | 43 | 2.6 | A128 | 4 | 0.2 |
| A14 | 30 | 1.8 | A120 | 4 | 0.2 |
| A196 | 24 | 1.5 | A107 | 3 | 0.1 |
| A145 | 19 | 1.1 | | | |

Table VIII: The accuracy (naive) for 205 authors, *i.e.*, the accuracy for each fold in the k-fold cross-validation (Challenge, Average, and Naive). All numbers are percentages.

| GCJ 2017 | | | GCJ 2018 | | | GCJ 2019 | | |
|----------|------|-----|----------|------|----|----------|------|------|
| C | 205 | N | C | 205 | N | C | 205 | N |
| C1 | 89.2 | ✓ | C1 | 70.2 | ✓ | C1 | 89.7 | ✗ |
| C2 | 87.8 | ✓ | C2 | 79.0 | ✗ | C2 | 88.7 | ✓ |
| C3 | 88.2 | ✓ | C3 | 87.3 | ✓ | C3 | 79.2 | ✗ |
| C4 | 90.2 | ✓ | C4 | 89.2 | ✗ | C4 | 88.7 | ✗ |
| C5 | 88.7 | ✓ | C5 | 79.0 | ✓ | C5 | 91.7 | ✗ |
| C6 | 93.6 | ✓ | C6 | 81.4 | ✗ | C6 | 86.3 | ✗ |
| C7 | 91.7 | ✓ | C7 | 70.2 | ✗ | C7 | 80.0 | ✓ |
| C8 | 90.2 | ✓ | C8 | 85.3 | ✓ | C8 | 79.5 | ✓ |
| A | 90.2 | 100 | A | 80.2 | 50 | A | 85.4 | 37.5 |

authorship to the code samples. To evaluate the impact of the feature-based approach on classification accuracy, we focused on assessing the accuracy of the target label as well. The target label refers to the specific label used to generate the set of ChatGPT-transformed code samples. By evaluating the accuracy of this target label, we were able to gain valuable insights into the effectiveness of the feature-based approach in improving the classification accuracy for both the ChatGPT and the non-ChatGPT codes.

We first obtained the baseline accuracies of the combined dataset, consisting of 205 authors, for each experiment. In the case of GCJ 2017, both the naive setting and the feature-based setting achieved the baseline accuracy of 90.2%, as presented in Table VIII and Table IX. As for GCJ 2018, the naive setting achieved the baseline accuracy of 80.2%, while the feature-based setting achieved 79.6%, as indicated in Table VIII and Table IX. For GCJ 2019, the naive setting reached an accuracy of 85.4%, while the feature-based setting achieved 85.2% as presented in Table VIII and Table IX. Building on these baselines, we can assess the models’ effectiveness in the context of code authorship attribution.

The initial experiment was performed using the GCJ 2017 dataset. When applying the naive approach, the classification accuracy reached 100%. Similarly, when utilizing the feature-based approach, both the target code and ChatGPT-transformed code achieved 100% accuracy as shown in Table VIII and Table IX. The reason behind this outcome can be attributed to the imbalanced usage of labels, where a single label was utilized in over 77% of the instances as we discussed in section VI-C. Consequently, even when employing the naive approach, it yielded the same result as the feature-based.

Subsequently, we proceeded with an experiment using the

Table IX: The accuracy (feature-based) for 205 authors, *i.e.*, achieved for each fold in the k-fold cross-validation process (Challenge, Average, Target, and Feature-based). All numbers are percentages.

| GCJ 2017 | | | | GCJ 2018 | | | | GCJ 2019 | | | |
|----------|------|-----|-----|----------|-----|------|--|----------|------|------|---|
| C | 205 | T | F | 205 | T | F | | 205 | T | F | |
| C1 | 90.2 | ✓ | ✓ | 67.3 | ✓ | ✓ | | 89.7 | ✓ | ✓ | ✓ |
| C2 | 88.7 | ✓ | ✓ | 78.5 | ✓ | ✓ | | 88.2 | ✓ | ✓ | ✗ |
| C3 | 86.8 | ✓ | ✓ | 86.8 | ✓ | ✓ | | 78.5 | ✗ | ✓ | ✓ |
| C4 | 90.2 | ✓ | ✓ | 89.7 | ✓ | ✓ | | 89.7 | ✓ | ✓ | ✓ |
| C5 | 89.2 | ✓ | ✓ | 77.0 | ✓ | ✓ | | 92.1 | ✓ | ✓ | ✗ |
| C6 | 91.2 | ✓ | ✓ | 82.9 | ✓ | ✓ | | 86.8 | ✗ | ✓ | ✓ |
| C7 | 91.7 | ✓ | ✓ | 71.7 | ✓ | ✗ | | 77.5 | ✗ | ✗ | ✗ |
| C8 | 93.6 | ✓ | ✓ | 83.4 | ✓ | ✓ | | 90.2 | ✓ | ✓ | ✓ |
| A | 90.2 | 100 | 100 | 79.6 | 100 | 87.5 | | 85.2 | 62.5 | 62.5 | |

Table X: Binary classification accuracy results for GCJ 2017, GCJ 2018, and GCJ 2019 for individual and combined training (Challenge and Average). All numbers are percentages.

| Individual | | | | Combined | | | |
|------------|------|------|------|----------|------|------|------|
| C | 2017 | 2018 | 2019 | 2017 | 2018 | 2019 | All |
| C1 | 96.3 | 74.5 | 96.8 | 97.5 | 86.0 | 91.8 | 91.8 |
| C2 | 99.8 | 82.8 | 98.3 | 99.8 | 88.8 | 95.5 | 94.7 |
| C3 | 93.8 | 97.3 | 93.0 | 94.8 | 96.3 | 85.8 | 92.3 |
| C4 | 92.5 | 77.0 | 88.0 | 96.5 | 91.5 | 88.3 | 92.1 |
| C5 | 87.5 | 97.8 | 99.0 | 94.0 | 91.5 | 98.3 | 94.6 |
| C6 | 98.3 | 97.3 | 84.0 | | | | |
| C7 | 81.3 | 99.3 | 93.3 | | | | |
| C8 | 77.8 | 91.8 | 98.5 | | | | |
| A | 90.9 | 89.7 | 93.8 | 95.5 | 90.8 | 91.9 | 93.1 |

GCJ 2018 dataset. In this case, we observed contrasting results between the naive and feature-based approaches. Due to the presence of multiple labels being used, the naive approach achieved a mere 50% accuracy as presented in Table VIII. However, the feature-based approach displayed a significantly higher accuracy of 87.5%, while still maintaining 100% accuracy of the target label as shown in Table IX. These findings suggest that the ChatGPT code authorship approach remains effective even when applied to ChatGPT-transformed code.

Lastly, we performed an experiment using the GCJ 2019 dataset, which yielded results similar to those of the GCJ 2018 experiment, with some minor differences. The naive approach encountered challenges in accurately classifying the authorship, achieving a classification accuracy of only 37.5% as shown in Table VIII. In contrast, the feature-based approach demonstrated improved accuracy compared to the naive approach, achieving a classification accuracy of 62.5%. However, it is worth noting that the accuracy of the target label decreased to 62.5% as presented in Table IX.

These findings confirm the effectiveness of the feature-based approach in attributing authorship to ChatGPT-transformed code. However, it is important to acknowledge that there may be a trade-off between achieving higher classification accuracy and maintaining accuracy of the target label.

E. Binary Classification

We conduct a binary classification experiment to compare the outputs of ChatGPT and human, focusing on these two classes. In the process, we created a combined dataset that includes data from three years dataset (GCJ 2017, 2018, and 2019). Subsequently, we tested this combined dataset alongside each individual year’s dataset. The results of the

binary classification for each individual year can be found in Table X, while the results for the combined dataset are presented in Table X. During the experiment with the combined dataset, we acquired both average challenge-level and average accuracy metrics, which can be found under the category labeled “All”. Furthermore, we computed challenge-level and average accuracy values for each individual dataset separately and presented them alongside their respective dataset names.

Overall, when we examine the outcomes of our experiments focused on binary classification accuracy, it becomes evident that the results consistently demonstrated impressively high levels of accuracy across various datasets. Specifically, the individual experiment involving the GCJ 2019 dataset yielded the most remarkable accuracy rate, standing at an impressive 93.8%. On the contrary, the lowest recorded accuracy, although still quite commendable, was 89.7%, which was observed in the context of the GCJ 2018 dataset. Moreover, the combined dataset exhibited an outstanding 93.1% accuracy, underscoring the robustness and reliability of our classification models across a wide range of challenges and styles.

F. Take Away

To explore ChatGPT’s code transformation capability and the effectiveness of the code authorship method, we conducted a series of experiments. Initially, we conducted experiments using the datasets of GCJ 2017 and 2018. However, we encountered inconsistencies in the results obtained from these experiments. Consequently, to further validate our findings and ensure consistency across different datasets, we conducted an additional experiment specifically focused on the GCJ 2019 dataset. This experiment was designed as a confirmation study to reinforce the reliability of our results and provide further evidence supporting our conclusions.

The observations derived from these experiments provided confirmation regarding the effectiveness of the ChatGPT code authorship approach when applied to ChatGPT-transformed code, specifically demonstrated through the GCJ 2018 and 2019 datasets. Additionally, the experiments revealed limitations in ChatGPT’s ability to produce a wide range of coding styles, with a maximum of 12 styles observed. Moreover, certain circumstances highlighted an imbalance in the usage of coding styles, such as the instance where a single label was employed in over 77% within the GCJ 2017 dataset.

G. Future Works

This study has primarily focused on evaluating ChatGPT’s capabilities in code transformation and authorship attribution. Thus it can be expanded on this foundation by exploring a broader range of LLMs, including state-of-the-art models such as Gemini-1.5-pro, GPT-4o, and Claude, as tools for code authorship attribution. These models, which have been trained on vast and diverse datasets including code, present promising potential for zero- and few-shot learning tasks. By analyzing their ability to attribute authorship accurately across multiple programming languages and coding styles, researchers can identify patterns and limitations in each model.

Additionally, further research could explore the impact of cross-language authorship attribution, where authorship is identified across code samples written in different programming languages. This is particularly relevant in understanding the robustness and generalization capabilities of LLMs in multilingual coding environments. Incorporating larger and more diverse datasets, such as those from open-source repositories or industry projects, could also provide deeper insights into the practical applicability of these methods.

VII. CONCLUSION

In this paper, we explored the transformative capabilities of ChatGPT and assessed the effectiveness of ChatGPT-based code authorship attribution. Our findings reveal that ChatGPT can effectively modify the stylistic features of code, presenting challenges for traditional authorship attribution methods that rely heavily on such features. Despite these transformations, ChatGPT demonstrates a limited capacity to produce only 12 distinct coding styles, with some styles being more frequently used than others. This limitation suggests that its transformations, while diverse, are not exhaustive. The results of our experiments highlight the resilience of ChatGPT-based authorship attribution methods, which maintained high effectiveness even when applied to ChatGPT-transformed code. In contrast, naive approaches struggled to achieve accurate attribution under the same conditions. Furthermore, a binary classification model trained on a combination of ChatGPT-transformed and non-ChatGPT code achieved an accuracy of up to 93%, underscoring its potential to reliably distinguish between these code types. These insights demonstrate both the capabilities and limitations of ChatGPT in code transformation and underscore the importance of robust attribution techniques to address the challenges posed by AI-generated code.

REFERENCES

- [1] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *CCS*. ACM, 2018.
- [2] M. Abuhamad, T. AbuHmed, D. Nyang, and D. Mohaisen. Multi- χ : Identifying multiple authors from source code files. *PoPETs*, 2020(3):25–41, 2020.
- [3] M. Abuhamad, J. Rhim, T. AbuHmed, S. Ullah, S. Kang, and D. Nyang. Code authorship identification using convolutional neural networks. *Future Gener. Comput. Syst.*, 95:104–115, 2019.
- [4] Amazon. Alexa - amazon. <https://alexa.amazon.com/>, 2023. Accessed: 2023-06.
- [5] B. A. Anders. Is using chatgpt cheating, plagiarism, both, neither, or forward thinking? *Patterns*, 4(3):100694, 2023.
- [6] R. Anil, S. Borgeaud, Y. Wu, J. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, D. Silver, S. Petrov, M. Johnson, I. Antonoglou, J. Schrittwieser, A. Glaese, J. Chen, E. Pitler, T. P. Lillicrap, A. Lazaridou, and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023.
- [7] Apple. Siri. <https://www.apple.com/siri/>, 2023. Accessed: 2023-06.
- [8] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *CCS*, pages 12–24. ACM, 2007.
- [9] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):14:1–14:39, 2010.
- [10] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game AI. In *AIIDE*. The AAAI Press, 2008.

- [11] S. Choi, R. Jang, D. Nyang, and D. Mohaisen. Untargeted code authorship evasion with seq2seq transformation. In *Computational Data and Social Networks - 12th International Conference, CSoNet 2023, Hanoi, Vietnam, December 11-13, 2023, Proceedings*, volume 14479 of *Lecture Notes in Computer Science*, pages 83–92. Springer, 2023.
- [12] S. Choi and D. Mohaisen. Attributing chatgpt-generated source codes. *IEEE Transactions on Dependable and Secure Computing*, 2025.
- [13] E. Choo, M. Nabeel, R. D. Silva, T. Yu, and I. Khalil. A large scale study and classification of virustotal reports on phishing and malware urls. *CoRR*, abs/2205.13155, 2022.
- [14] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. In *NeuRIPs*, 2017.
- [15] DeepMind. Deepmindalphacode. <https://alphacode.deepmind.com>, 2023. Accessed: 2023-06.
- [16] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *ACE*, pages 10–19. ACM, 2022.
- [17] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. PAL: program-aided language models. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023.
- [18] S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, 2011.
- [19] GitHubCopilot. Githubcopilot. <https://github.com/features/copilot>, 2023. Accessed: 2023-06.
- [20] Google. Google code jam. <https://codingcompetitions.withgoogle.com/codejam/archive>, 2023. Accessed: 2023-06.
- [21] Google Assistant. Google assistant, your own personal google default. <https://assistant.google.com/>, 2023. Accessed: 2023-06.
- [22] Google Gemini. Gemini - chat to supercharge your ideas - google. <https://gemini.google.com/>, 2023. Accessed: 2023-10.
- [23] M. Gupta, C. Akiri, K. Aryal, E. Parker, and L. Praharaj. From chatgpt to threatgpt: Impact of generative AI in cybersecurity and privacy. *IEEE Access*, 11:80218–80245, 2023.
- [24] S. Hobert. Say hello to ‘coding tutor’! design and evaluation of a chatbot-based learning system supporting students to learn to program. In *ICIS*. AIS, 2019.
- [25] M. Hoq, Y. Shi, J. Leinonen, D. Babalola, C. F. Lynch, T. W. Price, and B. Akram. Detecting chatgpt-generated code submissions in a CS1 course using machine learning models. In B. Stephenson, J. A. Stone, L. Battestilli, S. A. Rebelsky, and L. Shoop, editors, *Proceedings of the 55th ACM Technical Symposium on Computer Science Education, SIGCSE 2024, Volume 1, Portland, OR, USA, March 20-23, 2024*, pages 526–532. ACM, 2024.
- [26] A. C. Islam, R. E. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Security Symposium*, pages 255–270, 2015.
- [27] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhonova, and A. Matyukhina. Code authorship attribution: Methods and challenges. *ACM Comput. Surv.*, 52(1):3:1–3:36, 2019.
- [28] A. Kashefi and T. Mukerji. Chatgpt for programming numerical methods. *CoRR*, abs/2303.12093, 2023.
- [29] M. Kazemitabaar, R. Ye, X. Wang, A. Z. Henley, P. Denny, M. Craig, and T. Grossman. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In F. F. Mueller, P. Kyburz, J. R. Williamson, C. Sas, M. L. Wilson, P. O. T. Dugas, and I. Shklovski, editors, *Proceedings of the CHI Conference on Human Factors in Computing Systems, CHI 2024, Honolulu, HI, USA, May 11-16, 2024*, pages 650:1–650:20. ACM, 2024.
- [30] J. Kothari, M. Shevartlov, E. Stehle, and S. Mancoridis. A probabilistic approach to source code authorship identification. In *ITNG*, pages 243–248. IEEE, 2007.
- [31] M. Lajkó, V. Csuvik, and L. Vidács. Towards javascript program repair with generative pre-trained transformer (GPT-2). In *APR@ICSE*. IEEE, 2022.
- [32] Z. Li, Q. G. Chen, C. Chen, Y. Zou, and S. Xu. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *ICSE*, pages 1906–1918, 2022.
- [33] Z. Li, C. Wang, Z. Liu, H. Wang, S. Wang, and C. Gao. CCTEST: testing and repairing code completion systems. *CoRR*, abs/2208.08289, 2022.
- [34] Q. Liu, S. Ji, C. Liu, and C. Wu. A practical black-box attack on source code authorship identification classifiers. *IEEE TIFS*, 16:3620–3633, 2021.
- [35] B. D. Lund, T. Wang, N. R. Mannuru, B. Nie, S. Shimray, and Z. Wang. Chatgpt and a new academic reality: Artificial intelligence-written research papers and the ethics of the large language models in scholarly publishing. *J. Assoc. Inf. Sci. Technol.*, 74(5):570–581, 2023.
- [36] W. Ma, Y. Song, M. Xue, S. Wen, and Y. Xiang. The “code” of ethics: A holistic audit of AI code generators. *CoRR*, abs/2305.12747, 2023.
- [37] B. Marcus. Gphtreats-3: Is automatic malware generation a threat? 2023.
- [38] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers. Using an LLM to help with code understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 97:1–97:13. ACM, 2024.
- [39] N. Nguyen and S. Nadi. An empirical evaluation of github copilot’s code suggestions. In *MSR*, pages 1–5. ACM, 2022.
- [40] OpenAI. Chatgpt. <https://openai.com/blog/chatgpt>, 2023. Accessed: 2023-06.
- [41] OpenAI. Document of chatgpt. <https://platform.openai.com/docs/model-index-for-researchers>, 2023. Accessed: 2023-06.
- [42] OpenAICodex. Openaicodex. <https://openai.com/blog/openai-codex>, 2023. Accessed: 2023-06.
- [43] Y. M. P. Pa, S. Tanizaki, T. Kou, M. van Eeten, K. Yoshioka, and T. Matsumoto. An attacker’s dream? exploring the capabilities of chatgpt for developing malware. In *2023 Cyber Security Experimentation and Test Workshop, CSET 2023, Marina del Rey, CA, USA, August 7-8, 2023*, pages 10–18. ACM, 2023.
- [44] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot’s code contributions. In *S&P*, pages 754–768. IEEE, 2022.
- [45] S. Peleg and A. Rosenfeld. Breaking substitution ciphers using a relaxation algorithm. *Commun. ACM*, 22(11):598–605, 1979.
- [46] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. Do users write more insecure code with AI assistants? *CoRR*, abs/2211.03622, 2022.
- [47] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *USENIX Security*, pages 479–496, 2019.
- [48] H. Shahriar and M. Zulkernine. Information-theoretic detection of SQL injection attacks. In *HASE*, pages 40–47. IEEE, 2012.
- [49] W. Shahzad, A. B. Siddiqui, and F. A. Khan. Cryptanalysis of four-rounded DES using binary particleswarm optimization. In *GECCO*. ACM, 2009.
- [50] M. L. Siddiq and J. C. S. Santos. Generate and pray: Using SALLMS to evaluate the security of LLM generated code. *CoRR*, abs/2311.00889, 2023.
- [51] srcML. srcml v1.0.0. <https://www.srcml.org/>, 2023. Accessed: 2023-06.
- [52] O. SS, A.-K. AS, and A.-S. DM. Using genetic algorithm to break a mono-alphabetic substitution cipher. In *IOCS*, pages 63–67. IEEE, 2010.
- [53] B. Steven and T. S. MM. Source code authorship attribution using n-grams. In *Proceedings of the twelfth Australasian document computing symposium, Melbourne, Australia, RMIT University*, pages 32–39. Cite-seer, 2007.
- [54] N. Stiennon, L. Ouyang, J. Wu, D. M. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano. Learning to summarize with human feedback. In *NeurIPS*, 2020.
- [55] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan. Pythia: Ai-assisted code completion system. In A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, editors, *KDD*, pages 2727–2735. ACM, 2019.
- [56] H. Tian, W. Lu, T. O. Li, X. Tang, S. Cheung, J. Klein, and T. F. Bissyandé. Is chatgpt the ultimate programming assistant - how far is it? *CoRR*, abs/2304.11938, 2023.
- [57] A. Tlili, B. Shehata, M. A. Adarkwah, A. Bozkurt, D. T. Hickey, R. Huang, and B. Agyemang. What if the devil is my guardian angel: Chatgpt as a case study of using chatbots in education. *Smart Learn. Environ.*, 10(1):15, 2023.
- [58] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q. Han, and Y. Tang. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE CAA J. Autom. Sinica*, 10(5):1122–1136, 2023.
- [59] T. Ye, Y. Du, T. Ma, L. Wu, X. Zhang, S. Ji, and W. Wang. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. *arXiv preprint arXiv:2405.16133*, 2024.