Journal of Software Engineering and Applications, 2024, 17, 803-816
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

How Does Naming Affect Language Models on Code Analysis Tasks?

Zhilong Wang[1], Lan Zhang[2], Chen Cao[1], Nanqing Luo[1], Xinzhi Luo[3], Peng Liu[1]

[1]College of Information Science and Technology, The Pennsylvania State University, State College, USA
[2]School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, USA
[3]Heinz College, Carnegie Mellon University, Pittsburgh, USA
Email: zzw169@psu.edu

How to cite this paper: Wang, Z.L., Zhang, L., Cao, C., Luo, N.Q., Luo, X.Z. and Liu, P. (2024) How Does Naming Affect Language Models on Code Analysis Tasks? *Journal of Software Engineering and Applications*, **17**, 803-816.
https://doi.org/10.4236/jsea.2024.1711044

Received: August 30, 2024
Accepted: November 9, 2024
Published: November 12, 2024

Abstract

The Large Language Models (LLMs), such as GPT and BERT, were proposed for natural language processing (NLP) and have shown promising results as general-purpose language models. An increasing number of industry professionals and researchers are adopting LLMs for program analysis tasks. However, one significant difference between programming languages and natural languages is that a programmer has the flexibility to assign any names to variables, methods, and functions in the program, whereas a natural language writer does not. Intuitively, the quality of naming in a program affects the performance of LLMs in program analysis tasks. This paper investigates how naming affects LLMs on code analysis tasks. Specifically, we create a set of datasets with code containing nonsense or misleading names for variables, methods, and functions, respectively. We then use well-trained models (CodeBERT) to perform code analysis tasks on these datasets. The experimental results show that naming has a significant impact on the performance of code analysis tasks based on LLMs, indicating that code representation learning based on LLMs heavily relies on well-defined names in code. Additionally, we conduct a case study on some special code analysis tasks using GPT, providing further insights.

Keywords

LLMs, CodeBERT, Code Analysis

1. Introduction

Deep learning has demonstrated its significant learning ability in natural language processing (NLP). To deploy a natural language task, such as translation and text classification, researchers first pre-train a model to embed words into vectors using ELMo [1], GPT [2], and BERT [3]. These pre-trained models are initially trained on a large unsupervised text corpus and then fine-tuned on different downstream tasks. Such models are called large language models (LLMs) due to their relatively large number of model parameters. These LLMs have been deployed to the source code to learn program representations. Similar to natural language, the program representation learned from the source code using pre-trained models can be applied to several downstream program analysis tasks.

In 2020, Feng *et al.* proposed a pre-trained model called CodeBERT [4], based on Bidirectional Encoder Representations from Transformers (BERT), that learns general-purpose representations to support downstream NL-PL applications such as natural language code search, code documentation generation, and more. In 2021, Guo *et al.* proposed a new pre-trained model, GraphCodeBERT [5], which improves CodeBERT by enabling the model to capture more program semantic information, such as data flow. Recently, researchers have adopted ChatGPT for various code analysis tasks, such as code refinement [6], vulnerability discovery [7], and more. In industry, GitHub Copilot [8], powered by GPT, assists developers by suggesting code snippets, completing functions, and providing contextual code recommendations directly within Integrated Development Environments.

The inherent differences between natural language and programming language lead to challenges when applying NLP models to code analysis tasks. In natural language, words typically have a set, deterministic meaning within a given context, supported by established rules of grammar and usage. However, in programming languages, developers have the freedom to assign arbitrary strings as names for variables, methods, or functions. These names often do not follow any inherent meaning or convention and can be entirely context-dependent or even misleading. Unlike words in natural language, which contribute semantic value based on their definitions, names in programming languages are more symbolic and may not carry useful information about the code's function or behavior. As a result, LLMs trained on natural language may misinterpret these arbitrary names, especially if they overly rely on their literal meaning. This disconnect can lead to reduced accuracy and performance in code analysis tasks, where the model's reliance on learned natural language patterns may not transfer effectively to the abstract and flexible naming conventions used in programming.

Furthermore, a limited number of words are used in natural language, while in programming language, the number of words can be unlimited because a programmer can casually create a string to name a variable, regardless of whether the created string is interpretable. Therefore, it is doubtful whether the word embedding adopted in natural language is still efficient for solving program analysis tasks. If a model designer ignores the numerous differences between natural language and programming language and naively adopts methods from NLP, the designed model may suffer from the above limitations.

In this paper, we propose a taxonomy that groups features in programming languages into two categories: literal features and logical features. We aim to investigate how these different types of features affect the performance of current pre-trained models in downstream tasks and to what extent they influence the results. Specifically, to achieve this goal, we create eight types of datasets that mask off different types of features to measure their impact quantitatively. In each dataset, either variable names or method/function names are replaced with nonsense or misleading names. We then use well-trained models (CodeBERT) to perform two code analysis tasks on these created datasets and measure the impact of naming on the model's performance.

Based on the experimental results, we find that naming strategies have a significant impact on the performance of code analysis tasks based on LLMs, indicating that code representation learning heavily relies on well-defined names in code. Our experiments with CodeBERT reveal the extent to which literal and logical features impact performance. While these models exhibit remarkable capabilities, their effectiveness can be influenced by unreliable features in the analyzed code. Instances such as code generated from decompilation [9] or non-conventional code naming [10] might yield reduced accuracy, as LLMs' generalization ability is limited to the patterns and examples present in their training data.

Additionally, since ChatGPT is a conversational large language model that can provide intuitive feedback, we conduct a case study to investigate the pre-trained model's ability to comprehend programming language in specific settings.

2. Background

CodeBERT and GraphCodeBERT [4] [5] are based on the Transformer architecture and learn code representations through self-supervised training tasks, such as masked language modeling and structure-aware tasks, using a large-scale unlabeled corpus. Specifically, CodeBERT is pre-trained over six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go) and is designed to accomplish three main tasks: masked language modeling, code structure edge prediction, and representation alignment. By mastering these tasks, CodeBERT can comprehend and represent the underlying semantics of code, empowering security researchers with a powerful tool for source code analysis and protection. The pre-trained models have been applied to several downstream program analysis tasks: code search, clone detection, code translation, and code refinement.

ChatGPT [11], an AI chatbot developed by OpenAI and launched in November 2022, has garnered significant attention for its impressive capabilities. With the ability to respond to diverse prompts, ranging from answering questions in various fields to generating custom essays, people are truly astonished by its prowess. Researchers and users alike are intrigued by ChatGPT's potential to address questions across multiple domains, and they are keen to observe how its application may evolve in different areas. Recently, there has been research adopting ChatGPT for various code analysis tasks, such as code refinement [6], vulnerability discovery [7], and more.

3. Terminology: Literal Features and Logic Features

A source code file of a program consists of a sequence of tokens. These tokens can be grouped into three categories: keywords, operators (both defined in the programming language specification), and user-defined names.

Keywords are reserved words that have special meanings and purposes and can only be used for specific purposes. For example, for, if, and break are widely known keywords used in many programming languages. They are used in a program to change the control flow. A programming language usually contains only a limited number of keywords. For example, the C programming language contains 32 keywords, and Python 3.7 contains 35 keywords.

In addition to keywords, a programming language needs to define a set of operators. For example, arithmetic operators (e.g., +, (-), and *) and logical operators (e.g., and, or, and not) are two of the most important categories. Keywords and operators are defined by a programming language.

In addition, a programmer needs to define some tokens (i.e., names) to represent variables, structures, functions, methods, classes, and packages. When programmers write a code snippet, they can randomly choose any string to name these elements as long as the string does not conflict with reserved keywords and operators and follows the programming language specification. However, programmers have limited flexibility in choosing the keywords and operators. Only some keywords (such as for and while) and operators (such as ++, +1) are interchangeable. Thus, the program logic is fixed when using a set of keywords and operators in a program. In contrast, the names of variables, functions, or classes have no impact on the logic of the program.

Definition 1 (Literal Features and Logical Features): Given a piece of code, the **literal features** are features of the literal meaning of the variable names, function names, and programmer-readable annotation. These features could be removed without changing the functionality of the code. The **logical features** are features that control the program logic. The keywords and operators defined by a programming language specification are logical features.

4. Methodology

Currently, GraphCodeBERT takes code snippets of functions or class methods as data samples. It tokenizes keywords, operators, and developer-defined names from the code snippets and utilizes a Transformer to learn the data flow of the source code in the pre-training stage. Within a function or a method, developer-defined names can be grouped into three categories: 1) variable names, 2) method names, and 3) method invocation names. The program logic is not affected if we map these developer-defined names to other strings in the same namespace.

Since the pre-trained CodeBERT and GraphCodeBERT models, along with the datasets [12] for downstream tasks, are publicly available for download and use, we are able to conduct focused experiments to evaluate them. Here, we will conduct the evaluation based on two downstream tasks: natural language code search and clone detection.

The goal of code search is to find the code snippet that is most semantically relevant to a given natural language query from a collection of code. This reflects an LLM's ability to understand a specific function and map its intended purpose to the relevant code portion within the entire program.

Code clone detection aims to identify and measure the similarity between different code fragments, determining if they produce the same results when given the same input. This indicates whether an LLM's understanding has reached a level where it can mimic the code execution process and comprehend the logic flow.

We did not choose other downstream tasks because code refinement adopts an abstract representation and code translation involves two programming languages, which introduces challenges in uniformly anonymizing user-defined names. Since our experiments focus on the learning ability of pre-trained models on literal and other semantic features, and the downstream tasks are only used to evaluate the effectiveness of the pre-trained model, we believe that any reasonable downstream task should be appropriate.

4.1. Dataset

Then, we retrain the existing models and evaluate their performance on the two downstream tasks: natural language code search and clone detection. For the code search task, we used the test portion from the CodeSearchNet corpus collection [13]. This dataset, derived from publicly available open-source non-fork GitHub repositories and their associated documentation, provides code-text pairs for various code search challenges in both Java and Python. The testing dataset for the clone detection task is based on the BigCloneBench dataset [14], which provides a benchmark built from verified clones of ten common functionalities in different inter-project Java repositories.

Based on the datasets for these two tasks, we generate three types of datasets that anonymize three types of literal features in the code: variable names, method/function definition names, and method/function invocation names. Additionally, we create another type of dataset that anonymizes all three types of features together.

1) In the first type of datasets, we anonymize the variable names. An example is the change from it_end to var3 and finished to var4 between code:goodname and code:badname.

2) In the second type of datasets, we anonymize the method/function names. An example is the change from bubble_sort to fun1 between code:goodname and code:badname.

3) In the third type of datasets, we anonymize the method/function invocation names. An example is the change from pred to fun2 between code:goodname and

code:badname.

4) The last type of datasets are a combination of the three types of anonymization, which anonymize all three kinds of developer-defined names.

```
void bubble_sort(It begin, It end, Pred pred=Pred()){
    if ( std::distance( begin, end ) <= 1 ){ return; }
    auto it_end    = end;
    bool finished  = false;
    while ( !finished ){ //loop stop when no adjacent elements are
        not in required order.
        finished = true;
        std::advance( it_end, -1 );
        for (auto it = begin; it! = it_end; ++ it ){
            auto next = detail::advance( it, 1 );
            if (pred( * next, * it)){
                std::swap( * it, * next);
                finished = false;
            }
        }
    }
}
```

Code 1: A piece of code with meaningful variable/function names.

Besides, we adopt two strategies to anonymize the names: The first strategy, called "randomly-generated," randomly generates strings (e.g., "oe4yqk4cit2maq7t") without any literal meaning. The second strategy, called "shuffling," shuffles the names in the code. In this way, the shuffled names do not reflect the intention of the variables, functions, or invocations. For example, this strategy could replace "bubble_sort" with "aes_encryption".

```
void fun1(It var1, It var2, Pred fun2=Fun2()){
    if ( std::distance( var1, var2 ) <= 1 ){ return; }
    auto var3  = var2;
    bool var4  = false;
    while ( !var4 ){
        var4 = true;
        std::advance( var3, -1 );
        for (auto var5 = var1; var5! = var3; ++ var5 ){
            auto var6 = detail::advance( var5, 1 );
            if (fun2( * var6, * var5)){
                std::swap( * var5, * var6);
                var4 = false;
            }
        }
    }
}
```

Code 2: The same code without meaningful variable/function names.

Based on the four types of name sets to replace and the two replacing strategies, we eventually generated 16 variants of the original dataset from [7]. The details of the 16 datasets are shown in Table 1.

4.2. Experiment Design

Model Training and Fine-Tuning. We directly adopt the pre-trained model from GraphCodeBERT [5], which is pre-trained on three tasks. The first task involves

Table 1. Newly created dataset for our experiments.

	Task	Language	Var.[1]	Def.	Inv.	Anon.[2]	Size
$\mathcal{D}1$	Code Search	Java & Python	51	55	55	Rand.	175,878 & 266,738[3]
$\mathcal{D}2$	Code Search	Java & Python	51	55	55	Shuff.	175,878 & 266,738
$\mathcal{D}3$	Code Search	Java & Python	55	51	55	Rand.	175,878 & 266,738
$\mathcal{D}4$	Code Search	Java & Python	55	51	55	Shuff.	175,878 & 266,738
$\mathcal{D}5$	Code Search	Java & Python	55	55	51	Rand.	175,878 & 266,738
$\mathcal{D}6$	Code Search	Java & Python	55	55	51	Shuff.	175,878 & 266,738
$\mathcal{D}7$	Code Search	Java & Python	51	51	51	Rand.	175,878 & 266,738
$\mathcal{D}8$	Code Search	Java & Python	51	51	51	Shuff.	175,878 & 266,738
$\mathcal{D}9$	Clone Dete.	Java	51	55	55	Rand.	1316,444
$\mathcal{D}10$	Clone Dete.	Java	51	55	55	Shuff.	1316,444
$\mathcal{D}11$	Clone Dete.	Java	55	51	55	Rand.	1316,444
$\mathcal{D}12$	Clone Dete.	Java	55	51	55	Shuff.	1316,444
$\mathcal{D}13$	Clone Dete.	Java	55	55	51	Rand.	1316,444
$\mathcal{D}14$	Clone Dete.	Java	55	55	51	Shuff.	1316,444
$\mathcal{D}15$	Clone Dete.	Java	51	51	51	Rand.	1316,444
$\mathcal{D}16$	Clone Dete.	Java	51	51	51	Shuff.	1316,444

[1]**Var**. **Def.**, and **Inv**. represent that the variable names, method/function definition names, and the method/function invocation names are anonymized. [2]Anonymized methods: randomly-generated or shuffling. [3]Two numbers are the sizes of data samples corresponding two languages.

masked language modeling to learn representations from the source code. The second task is data flow edge prediction, aimed at learning representations from data flow. This is achieved by initially masking some variables' data flow edges and then having GraphCodeBERT predict those edges. The final task is variable alignment between the source code and data flow, which involves predicting the locations where a variable is identified to align representations between the source code and the data flow. GraphCodeBERT includes 12 layers of Transformer with 768-dimensional hidden states and 12 attention heads.

The pre-trained model is then fine-tuned on the downstream tasks on an NVIDIA DGX-1. Specifically, we train a model for each of the 16 datasets that we have crafted by randomly selecting 80% of the data samples as the training set and the remaining 20% as the test set.

Evaluation Matrix. We use Mean Reciprocal Rank (MRR) as our evaluation metric for the code search task, which assesses how well the model ranks the correct code snippet among 999 distractor snippets when given the documentation comment as a query. For the code clone detection task, we use the F1 score to measure the model's performance.

4.3. Experiment Results

Table 2 and Table 3 show the experiment results (MRR or F1 score) on the downstream tasks of code search and code clone detection, respectively. The second column shows the model performance reported in the original paper [5]. The third column indicates the anonymization methods adopted.

Table 2. Results on Code Search. The results are shown in $\frac{MRR}{Dataset}$ format.

Language	Orig.	Anon.	w/o Var.	w/o Def.	w/o Inv.	All
Java	70.36%	Random	$\frac{67.73\%}{\mathcal{D}1}$	$\frac{60.89\%}{\mathcal{D}3}$	$\frac{69.84\%}{\mathcal{D}5}$	$\frac{17.42}{\mathcal{D}7}$
		Shuffling	$\frac{67.14\%}{\mathcal{D}2}$	$\frac{58.36\%}{\mathcal{D}4}$	$\frac{69.84\%}{\mathcal{D}6}$	$\frac{17.03\%}{\mathcal{D}8}$
Python	68.17%	Random	$\frac{59.8\%}{\mathcal{D}1}$	$\frac{55.43\%}{\mathcal{D}3}$	$\frac{65.61\%}{\mathcal{D}5}$	$\frac{24.09\%}{\mathcal{D}7}$
		Shuffling	$\frac{59.78\%}{\mathcal{D}2}$	$\frac{55.65\%}{\mathcal{D}4}$	$\frac{65.61\%}{\mathcal{D}6}$	$\frac{23.73\%}{\mathcal{D}8}$

Table 3. Results on Clone Detection. The results are shown in $\frac{F1}{Dataset}$ format.

Language	Orig.	Anon.	w/o Var.	w/o Def.	w/o Inv.	All
Java	94.87%	Random	$\frac{92.64\%}{\mathcal{D}9}$	$\frac{93.97\%}{\mathcal{D}11}$	$\frac{94.72\%}{\mathcal{D}13}$	$\frac{86.77\%}{\mathcal{D}15}$
		Shuffling	$\frac{92.52\%}{\mathcal{D}10}$	$\frac{94.27\%}{\mathcal{D}12}$	$\frac{93.67\%}{\mathcal{D}14}$	$\frac{84.76\%}{\mathcal{D}16}$

The fourth, fifth, and sixth columns display the model performance when anonymizing different parts of the code: variable names, method/function definition names, and method/function invocation names, respectively. The last column shows the model performance when anonymizing all three types of names together. To associate the results with the corresponding datasets in Table 1, we provide not only the model performance but also the dataset numbers that were trained and tested on.

4.4. Analysis and Discussion

The results show that anonymizing variable names, method/function definition names, and method/function invocation names leads to a significant decline in model performance, regardless of whether we replace developer-defined names with "randomly-generated" strings or "shuffling" strings. Additionally, on average, datasets with "shuffling" strings perform worse than those with randomly-generated strings, suggesting that "shuffling" strings can mislead the models.

In terms of method/function names, anonymizing method/function definition names results in more significant degradation than anonymizing method/function invocation names, especially in code search tasks. This indicates that method/function definition names often strongly correlate with the overall function goal in most scenarios within the training dataset. Trained LLMs tend to infer a correlation between the method/function name and its actual purpose when tasked with code search. This predictability stems from open-source program datasets, which are well-organized and intended to be user-readable, unlike malware or commercial applications that obscure their true purpose from users. As for method/function invocation names, their impact is less pronounced because they operate within the function scope and influence only part of the function logic, similar to variable names. As shown in the results table, the score for changes in method/function invocation names is closer to changes in variable names than changes in method/function definition names.

As for different programming languages, we can discern the differences in degradation from the results of the code search task. Overall, if a single attribute is anonymized but the others remain unchanged, the degradation of LLM is more significant for Python than Java. However, if all three naming attributes in the code are changed, the Java task shows the most severe degradation, leading to an MRR score as low as 17.03. This disparity can be attributed to several factors related to language characteristics, naming conventions, and the overall ecosystem of the two languages. Java, as a statically typed language, provides additional context through type declarations, which compensates for the loss of meaningful names. In contrast, Python, a dynamically typed language, relies more heavily on descriptive variable and function names for context and clarity, aligning with its design philosophy of readability and simplicity. Developers expect meaningful names in Python to quickly understand code functionality.

Java code often includes more boilerplate and verbosity, which can provide additional contextual clues even if names are anonymized. Moreover, the Java ecosystem's strong typing and adherence to naming conventions in frameworks and libraries ensure consistency and maintainability in large codebases. These structural and syntactical elements help mitigate the impact of partial anonymization. However, modifying all names for both variables and method/functions in Java code disrupts these patterns, significantly impairing the LLM's ability to understand the code.

Similarly, in Python, anonymizing all three types of names led to a score as low as 23.73%, indicating that Python's reliance on descriptive naming for understanding is also critical. The combination of these factors underscores the challenges in adapting LLMs effectively to programming languages, necessitating models that can handle both literal and logical features to enhance performance in code analysis tasks.

Compared to the clone detection task, the performance degradation is more pronounced in the code search task. This difference arises from the inherent goals of the tasks: code search involves mapping a natural language description to a code block, while clone detection focuses solely on comparing code fragments.

In code search tasks, well-defined variable and method/function names are crucial because they help CodeBERT relate the code snippet to its descriptive purpose. Consequently, the model relies more on the natural language elements of the code and less on its underlying logic. Conversely, code cloning typically involves copying code with slight alterations, often in the names of variables or methods/functions, to avoid plagiarism or detection. Thus, detecting code clones requires ignoring these names and focusing on the actual logic flow of the code.

This fundamental difference between the two tasks explains the varying degrees of performance degradation observed in CodeBERT during code analysis. It also indicates that CodeBERT is well-trained for code clone detection tasks, recognizing that naming conventions have minimal impact on the actual logic of the code. Therefore, there is a growing need for pre-trained models specifically tailored to programming languages rather than natural language, to enhance their effectiveness in code analysis tasks.

Overall, our experiments reflect that current source-code level representation learning methods still largely rely on the *literal features* and ignore the *logical features*. However, the *literal features* are not always reliable, as mentioned in sec:intro. The current models still cannot effectively learn the logical features in the source code. Attackers can simply randomize the *literal features* to mislead the CodeBERT and GraphCodeBERT models.

5. Case Study

To verify our findings, we conducted case studies on ChatGPT to further investigate the details. ChatGPT is a state-of-the-art LLM designed to understand and generate human-like text based on the input it receives. Its ability to answer questions, provide explanations, and engage in conversations is helpful for us to understand the nuances of success or failure in our case study. Similar to CodeBERT, ChatGPT has also been adopted to perform source code level analysis and address security-related issues, such as vulnerability discovery and fixing [15] [16].

In this section, we conduct case studies on ChatGPT with a series of analytical tasks. Specifically, we first assess ChatGPT's capability to comprehend source code. Subsequently, we evaluate ChatGPT's proficiency in tackling specific security challenges and its generalization ability.

Semantic Comprehension

To evaluate whether ChatGPT has a better ability to understand the logical flows in the code without the influence of naming, we randomly select several code snippets from LeetCode[1], representing varying levels of difficulty, and assess ChatGPT's performance on the original code snippets and the corresponding obfuscated code snippets.

Our experiments show that ChatGPT can correctly understand the original code snippets. Taking the Supper Egg Drop problem (**Appx Q1**) as an example, it explains

[1]The source code is available at: https://github.com/Moirai7/ChatGPTforSourceCode.

the purpose of the code snippet, the logic and complexity of the algorithm, and provides suggestions to improve the code. Then, we manually obfuscated the code snippets by renaming the variables and function names and insert dummy code that will not have an impact on its results. The obfuscated code is shown in code:obfuscated. ChatGPT can not provide useful review of the obfuscated code (**Appx Q2**).

Based on the responses from ChatGPT, we can conjecture that ChatGPT learns the literal features of code snippets that exist within its training data and generates responses based on statistical patterns involving characters, keywords, and syntactic structures. This knowledge encompasses various aspects such as indentation, syntax, and common coding conventions. However, it lacks a profound comprehension of code semantics, logic, or context beyond the patterns and examples present in its training data. Consequently, it struggles to comprehend obfuscated code, as it relies heavily on literal features. This issue is exacerbated by the fact that ChatGPT is trained on a general corpus rather than a code-focused corpus like CodeBERT, making it more susceptible to misinterpretation due to name changes.

Observations: The conclusion of this case study aligns with the findings from our main experiments: users should expect degraded model performance when applying LLMs to analyze source code lacking sufficient information in variable, function, or class names. For instance, code generated from decompilation [11] or code that does not adhere to standard naming conventions [2] may yield lower accuracy, as demonstrated in questions **Appx Q1** and **Appx Q2**.

6. Conclusion

This study unveils the strengths and limitations of LLMs when applied to security-oriented code analysis tasks. Our focused experiments on CodeBERT and GraphCodeBERT shed light on the impact of literal and logic features on their performance, indicating the importance of considering the reliability of features in analyzed code. Users and researchers should be aware that LLM performance may be hindered when analyzing complex source code with insufficient information on variables, methods, and function names. We believe that our findings and analysis will provide valuable insights for future researchers in this domain, facilitating a deeper understanding of LLMs' potential and limitations in software security.

Acknowledgements

We are grateful to the anonymous reviewers for their useful comments and suggestions.

Funding

This work was supported by NSF CNS-2019340, NSF ECCS-2140175, and DOE CESER RC-40125b-2023.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Sarzynska-Wawer, J., Wawer, A., Pawlak, A., Szymanowska, J., Stefaniak, I., Jarkiewicz, M., *et al.* (2021) Detecting Formal Thought Disorder by Deep Contextualized Word Representations. *Psychiatry Research*, **304**, Article ID: 114135. https://doi.org/10.1016/j.psychres.2021.114135

[2] Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I. (2018) Improving Language Understanding by Generative Pre-Training.

[3] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2018) Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding.

[4] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., *et al.* (2020) Codebert: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics*: *EMNLP* 2020, November 2020, 1536-1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[5] Guo, D.Y., Ren, S., Lu, S., Feng, Z.Y., Tang, D.Y., Liu, S.J., Zhou, L. and Duan, N., *et al.* (2020) Graphcodebert: Pre-Training Code Representations with Data Flow.

[6] Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., *et al.* (2024) Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. *Proceedings of the IEEE*/*ACM* 46*th International Conference on Software Engineering*, Lisbon, 14-20 April 2024, 1-13. https://doi.org/10.1145/3597503.3623306

[7] Chen, C., Su, J.Z., Chen, J.C., Wang, Y.L., Bi, T.T., Wang, Y.L., *et al.* (2023) When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We?

[8] (2023) Github Copilot.

[9] Katz, D.S., Ruchti, J. and Schulte, E. (2018) Using Recurrent Neural Networks for Decompilation. 2018 *IEEE* 25*th International Conference on Software Analysis*, *Evolution and Reengineering* (*SANER*), Campobasso, 20-23 March 2018, 346-356. https://doi.org/10.1109/saner.2018.8330222

[10] Butler, S., Wermelinger, M. and Yu, Y. (2015) Investigating Naming Convention Adherence in Java References. 2015 *IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), Bremen, 29 September-1 October 2015, 41-50. https://doi.org/10.1109/icsm.2015.7332450

[11] OpenAI (2023) Introducing ChatGPT.

[12] Feng, Z.Y., Guo, D.Y., Tang, D.Y., Duan, N., Feng, X.C., Gong, M., *et al.* (2021) Code Pretraining Models.

[13] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M. and Brockschmidt, M. (2019) CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

[14] Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K. and Mia, M.M. (2014) Towards a Big Data Curated Benchmark of Inter-Project Code Clones. 2014 *IEEE International Conference on Software Maintenance and Evolution*, Victoria, 29 September-3 October 2014, 476-480. https://doi.org/10.1109/icsme.2014.77

[15] Surameery, N.M.S. and Shakor, M.Y. (2023) Use Chat GPT to Solve Programming Bugs. *International Journal of Information technology and Computer Engineering*, **3**, 17-22. https://doi.org/10.55529/ijitc.31.17.22

[16] Cheshkov, A., Zadorozhny, P. and Levichev, R. (2023) Evaluation of ChatGPT Model for Vulnerability Detection.

Appendix

Appx Q1: Could You Review the Bellowing Code?

```
int superEggDrop(int K, int N) {
    vector<vector<int>> dp(N + 1, vector<int>(K + 1, 0));
    int m = 0;
    while (dp[m][K] < N) {
        m++;
        for (int k = 1; k <= K; ++k)
            dp[m][k] = dp[m - 1][k - 1] + dp[m - 1][k] + 1;
    }
    return m;
}
```

Code 3: A solution to `Supper Egg Drop` problem from LeetCode.

Reply from ChatGPT: The code appears to be a dynamic programming solution to the "egg drop" problem, where one has K eggs and a building with N floors, and the objective is to find the minimum number of drops required to determine the floor from which the egg breaks.

The code creates a 2D vector dp with dimensions $(N+1)\times(K+1)$ and initializes all elements to 0. It then iterates through values of m (starting from 1) until the value of $dp[m][K]$ is greater than or equal to N. For each value of m, it updates the values of $dp[m][k]$ for each egg (k), based on the recurrence relation $dp[m][k] = dp[m-1][k-1] + dp[m-1][k] + 1$. Finally, it returns the value of m.

Overall, the code looks correct and efficient. It correctly computes the minimum number of drops required to determine the floor from which the egg breaks. The use of dynamic programming allows the algorithm to avoid unnecessary computations and runs in $O(K \times N)$ time, which is optimal for this problem.

However, it should be noted that this solution is not the only one to the egg drop problem, and there are other approaches that can be taken to solve it. Also, the code could benefit from some comments to explain the purpose and logic of the algorithm.

Appx Q2: Could You Review 4?

Reply from ChatGPT: The code you have provided appears to be written in C++ and defines a function called "gyjdfgadfsfvcvjhyt". Unfortunately, the function name and parameters do not give any indication of what the function is supposed to do. It would be helpful to have some context or explanation of the intended purpose of this function.

Regarding the code itself, there are some issues with readability and style that could be improved. For example, the use of single-letter variable names (*e.g.*, "kdsfjaldfie55qwer", "kdasd35sfja") can make it difficult for someone reading the code to understand what is going on. Additionally, the inconsistent use of whitespace and indentation can make the code more difficult to read.

It's also worth noting that the code appears to be using nested while loops, which can be difficult to follow and potentially lead to performance issues if the loops are executing a large number of times. Without more context, it's difficult to say whether this is a problem in this particular case.

Finally, there are no comments or documentation explaining what the code is doing, which could make it difficult for other developers to understand and modify the code in the future.

Overall, I would recommend improving the readability and style of the code, adding comments and documentation, and providing more context and explanation of the intended purpose of the function.

```
// obfuscated code snippet
int gyjdfgadfsfvcvjhyt(int kdsfjaldfie55qwer, int kdsfja) {
    vector<vector<int>> kdasd35sfja(kdsfja + 1, vector<int>(kdsfjaldfie55qwer + 1, 0));
    int kdasd35sfjada = 0;
    int af567856dfg = kdasd35sfja.size(), abf567856dfg = kdasd35sfja[0].size(),
        abf5678s56dfg = 0, abf5678sss56dfg = abf567856dfg - 1;
        int abfss556dfg =0;

    while (kdasd35sfja[kdasd35sfjada][kdsfjaldfie55qwer] < kdsfja) {
        while (abf5678s56dfg < af567856dfg && abf5678sss56dfg > -1) {
            int abf556dfg = kdasd35sfja[abf5678s56dfg][abf5678sss56dfg];
            if (abf556dfg == abfss556dfg) break;
            if (abfss556dfg > abf556dfg) abf5678s56dfg++;
            else abf5678sss56dfg--;
        }
        kdasd35sfjada++;
        for (int sdf23423 = 1; sdf23423 <= kdsfjaldfie55qwer; ++sdf23423)
            kdasd35sfja[kdasd35sfjada][sdf23423] = kdasd35sfja[kdasd35sfjada - 1][sdf23423 - 1] + kdasd35sfja[kdasd35sfjada - 1][sdf23423] + 1;
    }
    return kdasd35sfjada;
}
```

Code 4: **Obfuscated code based on Supper Egg Drop solution**. We renaming the variables and function names and insert dummy code that will not have a effect on its results.