Evaluation of Large Language Models on Code Obfuscation (Student Abstract)

Adrian Swindle[1], Derrick McNealy[2], Giri Krishnan[3], Ramyaa Ramyaa[4]
[1]Saint Louis University
[2] University of Southern Mississippi
[3]University of California, San Diego
[4]New Mexico Institute of Mining and Technology
adrian.swindle@slu.edu, derrick.mcnealy@usm.edu, gkrishnan@ucsd.edu, ramyaa.ramyaa@nmt.edu

Abstract

Obfuscation intends to decrease interpretability of code and identification of code behavior. Large Language Models (LLMs) have been proposed for code synthesis and code analysis. This paper attempts to understand how well LLMs can analyse code and identify code behavior. Specifically, this paper systematically evaluates several LLMs' capabilities to detect obfuscated code and identify behavior across a variety of obfuscation techniques with varying levels of complexity. LLMs proved to be better at detecting obfuscations that changed identifiers, even to misleading ones, compared to obfuscations involving code insertions (unused variables, as well as variables that replace constants with expressions that evaluate to those constants). Hardest to detect were obfuscations that layered multiple simple transformations. For these, only 20-40% of the LLMs' responses were correct. Adding misleading documentation was also successful in misleading LLMs. We provide all our code to replicate results at https://github.com/SwindleA/LLMCodeObfuscation. Overall, our results suggest a gap in LLMs' ability to understand code.

Introduction

Code obfuscations are functionality-preserving transformations that reduce the readability of the code rendering it harder to understand or reverse engineer (Martinelli et al. 2018). Obfuscated code remains a challenge for cybersecurity due to its ability to mask malware from conventional (signature based) detection methods(Martinelli et al. 2018). Further, ability of LLM to detect obfuscation in code reflect on their ability to identify equivalance of code functionality.

Large Language Models (LLMs) have been proposed for analyzing code and are also widely used in code synthesis(Austin et al. 2021; Chen et al. 2021; Li et al. 2022). Detecting obfuscation involves understanding code behavior and would be a specific test of reasoning and code analysis capabilities. Here, we study three LLMs (ChatGPT 3.5, Jurassic-2, and PaLM) with respect to their ability to detect obfuscation and to identify code behavior. Theoretically this problem, in general, is undecidable (Rice's Theorem). However, we are interested in very simple code which always terminates, whose behavior can be understood, and termination proved easily.

Method

ChatGPT 3.5 (OpenAI 2023), Jurassic-2 (AI21 2023), and PaLM (Google 2023) were chosen as the LLMs to evaluate, based on their robustness and performing well in preliminary tests. APIs for each of the LLM's was used in some form of Chat Completion. Default settings of parameters worked well in simple tests and were not changed.

We created base codes and obfuscations (described below) as we wanted to analyze the results based on the type of code and obfuscation on very simple codes and obfuscations. We used 21 distinct C++ base codes. All the base codes used compute simple functions, such as those that would be used in an introductory programming course. Complexity varied from printing integers 1 to 10 each in a new line, to checking whether the input is prime. Data structures and controls structures (code with and without loops and recursion) used were also varied to ensure that the LLMs are tested against a wide spectrum of coding tasks. We also included pieces of code whose behavior was simple but uncommon (checking whether input excluding the letter 'x' is a palindrome, printing a space followed by 6 newlines, etc.). These pieces of code were included to lower the likelihood of the LLMs having encountered them during training.

We used obfuscations with varying complexity. They can be grouped as (i) obfuscations that do not change the abstract syntax tree: These include transformations such as removing spaces and new lines, changing identifier names (shuffling the identifiers already used in the base code, using random identifiers, using misleading identifiers etc.), changing strings to ASCII etc. One obfuscation of note inserted misleading documentation. (ii) Obfuscations that change the abstract syntax tree: These include transformations like inserting unused variables, unnecessary statements (if-then statements, for-loops), changing math constants with complex expressions that evaluate to constants, replacing for-loops with recursion, etc. (iii) Layered obfuscations that combined multiple transformations.

The LLMs were tested using 3 prompts : (i) "Do these pieces of code achieve the same goal?" (ii) "Is the functionality of these pieces of code the same?" (iii) "What does this piece of code do?". Prompts 1 and 2 included the obfuscated and original code. The words "goal" and "functionality" in prompts 1 and 2 respectively, are the key differences between the prompts. Prompting the LLM with "goal" was

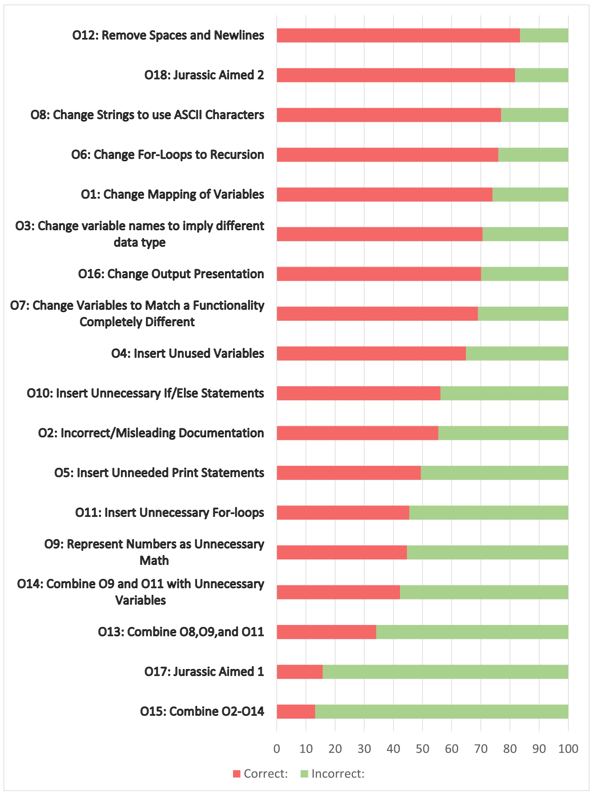

Figure 1: Average accuracy of LLMs across different obfuscation methods.

aimed at leading it towards giving an answer regarding the outcome of the code. "Functionality" was used in hopes of leading the LLM to analyze how the code functions instead of the outcome. Prompt 3 differed from 1 and 2 because it does not include the original code. The prompt is purposefully vague to see how well the LLM understands the code without any context.

We used the following accuracy measure: if the LLM correctly answered (that the codes were obfuscations of each other/have the same functionality for prompts 1/2 or gave the correct functionality for prompt 3) and gave the correct explanation of its answer, then we marked it as a correct response. If either the answer or the explanation of the answer is incorrect, the response is marked as incorrect. This was because internal inconsistencies of the response (final answer and explanation) indicate lack of understanding. We discarded cases where the LLM produced an error and did not produce a response. Additional details of methods and analysis are provided in our Github repository [1].

Results

When comparing between the LLMs, Jurassic was found to be the best at identifying the true functionality of a script. Jurassic correctly answered the prompt and properly explained the code in 60% of its responses. PaLM and ChatGPT have a similar rate of correct responses (47% and 47.45%).

Base codes that were more complex were harder to decipher, as expected, as were the base codes that had no purpose (and were unlikely to have been seen by the LLMs).

The worst accuracy were on obfuscations that layered transformations. Following such obfuscations, in general, were obfuscations that change the abstract syntax tree fooled the LLMs more than the ones that do not. There were 2 notable exceptions to this (i) LLMs were not fooled by the obfuscation which transformed for-loops into recursive code. We believe this is because the function name was descriptive, and the code itself was simple enough that recursive and for-loop versions may have been used to train the LLMs. (ii) LLMs were fooled by inserting misleading documentation.

Across all LLMs, prompting the LLM to give the functionality of the obfuscated code(prompt 3) gave the highest accuracy (˜55%). While the prompt for asking if the original and obfuscated code had the same functionality(prompt 2) gave the lowest accuracy (˜48%). It appears that the inclusion of the original code, actually impaired the LLMs ability to understand the codes as a whole. Due to the vagueness of the prompts and a 7% range, it is unclear how significant the differences in the prompts are. It would be natural to assume that the performance of the LLMs will be better given more context through the prompts, but it is possible that more context could cause more error.

Discussion

In this work, we examined various obfuscation methods on a variety of code on various LLMs. The success of an obfuscation was determined by how many layers of obfuscation it contained and the type of obfuscation. The top 4 most successful (O13, O14, O15, O17) all used a layered approach to their obfuscations, with O18 being the most unsuccessful obfuscation. O15 was the most successful because the goal of the obfuscation is to take all the previous methods and apply it to the code. Based on all the obfuscations where the LLMs failed, or succeeded, Jurassic and PaLM relied heavily on variable names to identify the code. This can be seen in cases where the code contains complex math. Jurassic and PaLM only use the variable name to understand the function of the variable without deciphering the complex math. This is where ChatGPT fails.

Another observation from our obfuscations is LLMs can accurately detect O18, which attempts to confuse the LLMs by having descriptive comments for the original code spread throughout the obfuscated code. Even though the comments have no relevance to the obfuscated code, they still detail what the code should accomplish, providing the LM with all the information it needs to understand the code.

In conclusion, the LLMs were not able to understand the functionality of obfuscated code. Transformations that included changes to the abstract syntax tree and layered multiple types of transformations had the lowest accuracy for detecting obfuscation and code functionality by the LLMs.

[1]https://github.com/SwindleA/LLMCodeObfuscation

References

AI21. 2023. Jurassic-2 models. https://docs.ai21.com/docs/jurassic-2-models#jurassic-2-ultra-unmatched-quality. Accessed: 2023-07-14.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Google. 2023. API Documentation. https://developers.generativeai.google/api. Accessed: 2023-07-14.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Sangaiah, A. K.; and Cimitile, A. 2018. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119.

OpenAI. 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5. Accessed: 2023-07-14.