LLM Performance in Code Obfuscation: Insights from Gemini and Llama3

Carlo Velarde

University of Iowa, carlo-velarde@uiowa.edu

Caleb Parten

Eastern New Mexico University, caleb.parten@enmu.edu

Ramyaa Ramyaa

New Mexico Institute of Mining and Technology, ramyaa.ramyaa@nmt.edu

This paper investigates the performance of large language models (LLMs) in code obfuscation, focusing on Gemini and Llama3. Using over 500 unique JavaScript code snippets, we evaluated obfuscation accuracy and embedding quality. Employing cosine similarity algorithms, we assessed the proximity of embeddings for obfuscated and non-obfuscated code. Our findings suggest that embeddings are more influenced by code similarity than functionality, based on trends observed using the Levenshtein distance algorithm. Gemini produced higher quality embeddings and specific obfuscations but had more errors. Meanwhile, Llama3 had fewer errors but lacked obfuscation specificity. These insights highlight the strengths and limitations of current LLMs in code obfuscation, offering a foundation for future research in software security.

CCS CONCEPTS • Security and privacy~Software and application security~Software reverse engineering • Computing methodologies~Artificial intelligence~Natural language processing

1. Introduction

Code obfuscation is crucial for enhancing software security by making code difficult to understand and reverse engineer. Traditional obfuscation methods, such as renaming identifiers and altering control flow, are manual and labor-intensive, prompting the need for automated solutions.

Recent advancements in artificial intelligence have introduced large language models (LLMs) as promising tools for automating code obfuscation. Models like Gemini and Llama3, trained on vast amounts of code, can generate obfuscated code snippets with minimal human intervention. However, their effectiveness and reliability in producing high-quality obfuscations are underexplored.

This paper investigates the performance of Gemini and Llama3 in code obfuscation using over 500 unique JavaScript code snippets. We assess the accuracy of obfuscations and the quality of embeddings produced by these models. By employing similarity algorithms, we measure the proximity of embeddings for obfuscated and non-obfuscated code, providing insights into the semantic integrity of the code.

Our findings reveal that Gemini produces higher quality embeddings and specific obfuscations but has higher error rates, while Llama3 demonstrates fewer errors but lacks the ability to implement specific obfuscation methods. Additionally, we investigate whether embeddings are influenced more by textual similarity or functionality. Using the Levenshtein distance algorithm, we compare the string similarity of obfuscated and non-obfuscated code and analyze its relationship with the cosine similarity of the embeddings These insights highlight the strengths and limitations of current LLMs in code obfuscation, laying the groundwork for future research in software security.

* 1. Obfuscation Methods

Code obfuscation enhances software security by making code difficult to understand and reverse engineer. Popular techniques include naming obfuscation, dead code insertion, and control flow alteration. Naming obfuscation changes identifiers to meaningless names. Dead code adds non-functional code to increase complexity. Control flow transforms simple control structures into convoluted ones which maintains behavior but complicates the logic.

Traditionally manual and labor-intensive, these methods can be automated using large language models (LLMs) like Gemini and Llama3. Our research evaluates LLMs performance at obfuscating over 500 unique and simple JavaScript code snippets. We check for two features of the LLM’s obfuscated code: output and implementation. We check whether the obfuscated code has the same output as the normal code and we check whether the LLM implemented the specified obfuscation method. See table 1 for example of data collected.

Table 1: Sample representing collected data from LLM obfuscations

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| code | output | obfuscated\_code | obfuscated\_output | type | equal | implementation |
| const kilometers =10;  const miles = kilometers \* 0.621371;  console.log(`10 kilometers is ${miles} miles`); | 10 kilometers is 6.21371 miles | const kilometers=10;const foo='bar';const miles=kilometers\*0.621371;const baz='qux';console.log(`10 kilometers is ${miles} miles`); | 10 kilometers is 6.21371 miles | Dead Code Obfuscation | True | True |

* 1. Testing Obfuscation Methods

After running both Gemini and Llama3 on 500 JavaScript code snippets, we found great differences between the two. We manually tested how many obfuscated snippets had the correct obfuscation implementation, the correct output, and both the correct obfuscation and output. Correct implementation was graded on whether the LLM was able to implement the specified obfuscation method. If we asked for naming obfuscation, and the results were dead code obfuscation, then it is considered a failure. We also considered it a failure if we asked specifically for dead code obfuscation and it returns code with dead code and naming obfuscations.

Regarding Gemini’s results in obfuscation implementation, 332 passed and 168 failed which is around a 66% success rate. Gemini’s results in correct output were 296 passed and 204 fails which is around a 59% success rate. Gemini’s results in both correct implementation and output were 208 passed and 292 failed which is around a 42% success rate. Gemini had best success with naming obfuscation and lowest success in dead code obfuscation.

Regarding Llama3’s results in obfuscation implementation, 117 passed and 383 failed, which is around a 23% success rate. Llama’s results in correct output were 436 passed and 64 failed which is around an 87% success rate. Llama’s results in both correct implementation and output were 101 passed and 399 failed which is around a 20% success rate. Llama largely failed in providing the correct obfuscation method. It struggled to single out obfuscation methods. If it obfuscated code, it would use all methods, despite being asked to only use one specific method. However, it had great success in code output. It would consistently provide the correct output.

1. Using embeddings

We investigated whether LLM’s can find similarity between code snippets, especially between corresponding obfuscations. The goal was to get an idea on if LLM’s can take obfuscated code as input and either return a non-obfuscated version of the code or explain its functionality. To determine whether LLM’s can find similarity between code snippets, we looked into their embeddings.

Embeddings are multi-dimensional vectors that represent the semantic meaning of given tokens. These vectors are learned during the training process of the model. Each number in the vector represents a weight.

A diagram of embedding model

Description automatically generated

Figure 1: Example of embedding process. Model by OnFinance AI via LinkedInn, (https://www.linkedin.com/pulse/how-do-embeddings-work-large-language-model-llm-onfinanceofficial-54awc/)

* 1. Similarity Analysis

Using our table of 500 normal and obfuscated code snippets, we generated embeddings using both Gemini and Llama3 models. We generated embeddings for the 500 normal and 500 obfuscated code snippets. We then stored them in a data set (*see table 1*). Now that we had the embeddings, we want to determine their proximity in the latent space. The latent space is a multi-dimensional vector space where each element represents an embedded input. Embeddings with high semantic similarity are “closer” together whereas low semantic similarity are “farther” together.

We wanted to test whether the normal code and corresponding obfuscated code would be close together in the latent space. In the perfect scenario, the LLM would be able to determine that the code snippets are functionally equivalent and therefore would give them similar embeddings. To find out their proximity in the latent space, we used the cosine similarity algorithm. This algorithm returns a number between -1 and 1 where -1 is not similar and 1 is very similar.

* 1. Similarity Results

After applying the cosine similarity algorithm to each pair of corresponding embeddings, we discovered that the majority of the embedding pairs had a high similarity rating. Most of the Gemini embeddings were .9 or higher and the majority of the Llama3 embeddings were .7 or higher. These findings give ground to the assumption that Gemini is better than Llama3 at finding similarities between codes.

A graph with different colored bars

Description automatically generated

Figure 2: Histograph showing the similarity scores of 500 obfuscated and non-obfuscated code snippet embeddings.

* 1. Similarity Relations

As we manually went over the table, we found a trend where higher textual similarity (where the code has similar structure and keywords) seemed to relate to a higher cosine similarity score. To quantify this, we needed to convert the code snippets intro string and perform a string similarity algorithm. The idea is to find how similar both strings are to each other. We chose to use the Lavenshtein distance algorithm. This algorithm returns the number of single-character insertions, deletions, or edits required in order to transform one string to the other. We applied the algorithm to the string representations of the code. We then took the edit distance and applied a simple formula that turned the edit distance into a similarity score where 0 is no similarity and 1 is max similarity. In the formula below, distance is the edit distance and max length is the length of the larger string.

After applying the Lavenshtein distance algorithm to each pair of code snippets, we graphed the results. We believe the graph shows a close enough correlation between textual similarity and cosine similarity that warrants future research into this topic. Looking at figure 3, as the string similarity increases, the cosine similarity also increases. The lowest cosine similarity scores correspond to the lowest string similarity scores.

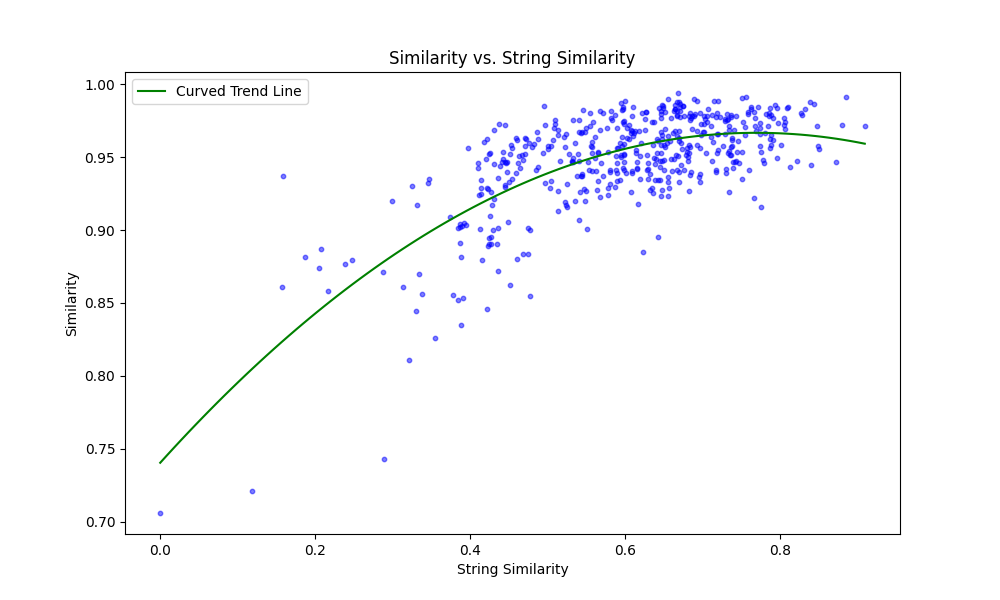


Figure 3: Scatter plot representing the relation between string similarity and cosine similarity.

1. Future research

During our research, we came across hints which lead us to believe that current LLM’s create code embeddings based on textural similarity rather than functionality. We also showed that LLM’s often fail in performing correct obfuscations despite the use of simple code snippets and advanced prompting techniques. This causes inadequate and limited obfuscations. The issue stems from the inability to give additional parameters when generating embeddings. Embeddings only take in the input sequence, nothing else. Therefore, this causes future research in how to generate more accurate embeddings in the context of obfuscation. The next step would be to fine-tune or train the models on obfuscation data.

1. Conclusion

In this study, we evaluated the performance of large language models (LLMs) Gemini and Llama3 in code obfuscation, using over 500 unique JavaScript code snippets. We examined their ability to implement specific obfuscation methods while maintaining the functionality of the code. Our findings revealed that while Gemini produced higher quality embeddings and could differentiate between obfuscation techniques, it had a higher error rate in terms of code execution. On the other hand, Llama3 had fewer errors in code execution but struggled with consistently applying specific obfuscation methods.

Our similarity analysis using the cosine similarity and Levenshtein distance algorithms indicated that embeddings generated by Gemini and Llama3 are influenced more by textual similarity than by the functional equivalence of the code snippets. This insight suggests that further refinement is needed in LLM training to better capture functional similarities rather than just textual ones.

Our findings highlight both the potential and limitations of using LLMs for automated code obfuscation. The ability of Gemini to perform specified obfuscation methods shows promise; however, Llama3's lack of correct implementation underscores its reliability. Moreover, the potential tendency of both models to prioritize textual similarity over functionality points to a critical area for future research. Advancements in this field could lead to more robust and secure automated obfuscation tools, which will overall enhance software security.

ACKNOWLEDGMENTS

This research was hosted at New Mexico Tech during May-August 2024. The program was sponsored by the National Science Foundation under the Research Experience for Undergraduates program. Grant sponsor: National Science Foundation and Grant no: CRP 10-2012-03.