CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

On Obfuscating JavaScript Code Using Large Language Models

Master's Thesis in Computer science and engineering

Andreea-Ioana Cîrstoiu
Siyu Heng

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

On Obfuscating JavaScript Code Using Large Language Models

Andreea-Ioana Cîrstoiu
Siyu Heng

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

On Obfuscating JavaScript Code Using Large Language Models
Andreea-Ioana Cîrstoiu
Siyu Heng

Supervisor: Philipp Leitner, Department of Computer Science and Engineering
Examiner: Daniel Strüber, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Andreea-Ioana Cîrstoiu
Siyu Heng
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Large Language Models (LLMs) have become increasingly popular for their proven capabilities in code analysis and synthesis, and code obfuscation is a suitable task. Generally, the purpose of obfuscation is to make a program difficult to understand. More specifically, it is widely applied to JavaScript code, as it is a popular language used to build client-side web applications. One reason to obfuscate JavaScript code is to prevent anyone from copying proprietary work. Code obfuscation is widely applied and studied in the context of cybersecurity, particularly in relation to malware or intellectual property protection.
Existing obfuscators have been built to implement obfuscation patterns of ranging complexities. Research in the area of code obfuscation using LLMs has emerged in recent years and is continually evolving. A potential gap in research is whether LLMs can obfuscate code using patterns that the current standard deobfuscators cannot reverse-engineer, and whether it is possible for the LLMs to replace existing obfuscators. To achieve this goal, it is essential to investigate whether and how LLMs can apply obfuscation transformations to code, as well as the impact that prompt engineering techniques have on the results.
In this laboratory experiment, our goal is to determine the extent to which an LLM can obfuscate JavaScript code. We choose an open-weight LLM and we craft prompts to obfuscate standalone, relatively simple code snippets. A key component of our work is a dedicated, free-to-use obfuscation tool that serves as our baseline for evaluating the LLM results. We prompt the model iteratively, then, using data visualization and descriptive statistics, we analyze and interpret the results. Our results show that the chosen LLM can obfuscate simple JavaScript code; however, the choice of prompt engineering technique is crucial. Some LLM-obfuscated code snippets differ significantly from the original code, while maintaining the original behavior. However, the LLM also yields obfuscated code that changes the original behavior, has errors, or is significantly similar to the original code.

Keywords: obfuscation, Large Language Model (LLM), JavaScript, prompt engineering, software engineering

Acknowledgements

We want to express our gratitude to our supervisor, Philipp Leitner, and our examiner, Daniel Strüber, for their support and guidance during our Master's thesis project. We also want to thank Nicole Andrea Quinstedt and Francisco Gomes de Oliveira Neto for their support in working with computing resources, as well as guidance on data analysis and visualization.

Computing resources were provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council (Vetenskapsrådet) under contract 2022-06725.

Andreea-Ioana Cîrstoiu & Siyu Heng, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

Generic code analysis and processing tools continue to improve, and large language models (LLMs) have gained popularity for their strong capabilities across various tasks, including code analysis, code generation, and human interaction [1]. They can generate various outputs based on the data they were already trained on, as stated by Martineau [2]. The model can be customized to perform tasks by carefully engineering prompts.

Obfuscation aims to make a program difficult to understand and is widely used in JavaScript client-side web applications for security-related purposes. Although it is not recommended as the sole security measure, it helps hide details to protect proprietary software or malicious code. Various code transformations can be applied to the code, which makes obfuscation a suitable task for LLMs. Research on both code obfuscation and deobfuscation is extensive, encompassing studies on various programming languages and various purposes. Standard deobfuscators can reverse-engineer most, if not all, JavaScript obfuscations. The novelty of LLMs can bring new insights into obfuscation patterns, but further work is needed to determine if LLMs are suitable for inclusion in standard daily obfuscation pipelines and whether they can replace existing standard obfuscators, eventually leading to the development of new transformations that are more resilient to reverse-engineering.

This study aims to explore the extent to which LLMs can obfuscate JavaScript code by carefully crafting prompts, enabling the adaptability of LLMs, and analyzing the results. Its focus is not on cybersecurity, but instead on whether the LLM can apply obfuscation transformations to JavaScript code and whether the results differ significantly from those of a baseline obfuscation tool, JavaScript Obfuscator [3].

1.1 Problem Statement

Code obfuscation has been widely analyzed and reviewed from the perspective of the cybersecurity domain. Whether the focus is on malware or possible proprietary software components, the principle of applying code transformations to obfuscate code is the same. Obfuscation makes the code difficult to read and understand.

JavaScript is a popular scripting language used in web development. Applications

using JavaScript distribute their code in source form. As a result, the code is directly visible to anyone accessing a browser or app. Therefore, potential attackers can easily attempt to find vulnerabilities in the code and exploit them to their advantage [4]. Although obfuscation is an application of security through obscurity and should never be used as the sole security measure, it helps hide details and delay potential attacks.

JavaScript code is often obfuscated for various purposes. Many free-to-use and proprietary tools are available online to perform these operations or integrate into projects or pipelines. LLMs are gaining popularity in related research. Still, further studies are needed to learn whether and how LLMs can apply different known or new patterns to the code and how the results can vary based on the prompt phrasing.

In this thesis, we investigate several prompt engineering techniques and select one to build an optimal prompt that harnesses the capabilities of LLMs for obfuscating JavaScript. We also establish a baseline obfuscation tool dedicated to evaluating the LLM results.

1.2 Research Questions

We divide our research goal of determining to what extent LLMs can obfuscate JavaScript code into the following research questions:

RQ 1: To what extent does the LLM-obfuscated code maintain the original behavior?

RQ 2: Is the LLM-obfuscated code significantly different compared to the original code?

1.3 Significance of the Study

This study aims to determine the extent to which an LLM can perform obfuscation code transformations in comparison to JavaScript Obfuscator [3]. The prompt engineering technique of choice is crucial in the interaction with the model. The model must only rely on the instructions provided in the prompt we build and not refer to any other tools and resources available online.

This thesis presents a quantitative analysis of code transformations using an LLM in the given context, compared to the JavaScript Obfuscator. It also includes an analysis of prompt engineering techniques and provides insights into our chosen prompt engineering approach. Our study identifies potential errors that may arise when obfuscating code using an LLM. Our work is intended to benefit both researchers and practitioners by providing a starting point for code analysis and processing using LLMs, which can be further explored through more complex code obfuscation transformations.

1.4 Thesis Outline

Chapter 2 summarizes the related work in the field, followed by our methodology presented in Chapter 3. Then, the results of our research are presented in Chapter 4. In Chapter 5, which summarizes the discussions, we give a few details on how the experiment was conducted, some challenges we encountered, and the threats to validity we identified. The study ends with the conclusions in Chapter 6.

2

Related Work

2.1 Code Obfuscation

Obfuscation transformations can be categorized into layout transformations, control flow transformations, and data abstraction. Layout transformations usually involve removing comments and formatting. Control flow transformations include dead code injection, reordering, and redundant computation. At the same time, data abstraction can be achieved by name modifications, updating inheritance relations, or changing the structure of data arrays [5, 6].

The following examples illustrate individual obfuscation transformations. However, dedicated tools combine several types of transformations, making it significantly challenging to reverse engineer the result without an automated deobfuscator.

2.1.1 Code Obfuscation Techniques by Example

2.1.1.1 Layout Transformations

The following snippet is obfuscated by renaming identifiers using random hexadecimal substitutes.

```
1 function sumNumbers(num1, num2) {
2     // calculates the sum of numbers
3     const sum = num1 + num2;
4     return sum;
5 }
6 sumNumbers();
```

Listing 2.1: Original Code - Layout transformations

```
1 function _0x141f18(_0x359fc4, _0x9675d) {
2     const _0xfd2aa1 = _0x359fc4 + _0x9675d;
3     return _0xfd2aa1;
4 }
5 _0x141f18();
```

Listing 2.2: Obfuscated Code - Layout transformations

2.1.1.2 Control Flow Transformations

The following example implements dead code injection, amongst other transformations such as renaming identifiers or removing string literals and placing them in a special array. Dead code is a segment of unnecessary code that is meant to confuse, mislead, or throw off any reverse-engineering attempts.

```
function sumNumbers(num1, num2) {
  return "Hello world" + (num1+num2);
}
sumNumbers();
```

Listing 2.3: Original Code - Control flow transformations

```
function _0x474a6b(_0x52f9e3, _0x5a0bb5) {
    var _0x471264 = _0x572f;
    return _0x471264(0x0) + (_0x52f9e3 + _0x5a0bb5);
}
function _0x572f(_0x312e67, _0x572f0d) {
    var _0x59b378 = _0x312e();
    _0x572f = function (_0x4e86de, _0x765e2b) {
        _0x4e86de = _0x4e86de - 0x0;
        var _0x31be66 = _0x59b378[_0x4e86de];
        return _0x31be66;
    };
    return _0x572f(_0x312e67, _0x572f0d);
}
function _0x312e() {
    var _0x4ac9c1 = ['Hello\x20world'];
    _0x312e = function () {
        return _0x4ac9c1;
    };
    return _0x312e();
}
_0x474a6b();
```

Listing 2.4: Obfuscated Code - Control flow transformations

2.1.1.3 Data Abstraction

In the following example, literal strings are split into chunks of a specified length. The value used in this example for the length is 2.

```
function example() {
    return 'abcdefg';
}
example();
```

Listing 2.5: Original Code - Data abstraction transformations

```
function _0x2e4208(){return'ab'+'cd'+'ef'+'g';}
_0x2e4208();
```

Listing 2.6: Obfuscated Code - Data abstraction transformations

2.2 Prompt Engineering

Prompt engineering has become an essential component in the artificial intelligence domain by extending the capabilities of large language models (LLMs). Prompts leverage task-specific instructions that guide the model output without altering its parameters.

In their survey paper, Sahoo et al. [7] categorize 29 distinct prompt engineering techniques, out of which the zero-shot, few-shot, and Chain-of-Thought (CoT) are suitable for building new tasks without training data, such as code obfuscation and deobfuscation. Crafting prompts for novel tasks requires trial-and-error experimentation, and different prompt templates with varying wording choices can lead to significant differences in accuracy [8].

Zero-shot prompting was introduced by Radford et al. [9]. It removes the need for extensive training data, relying on carefully crafted prompts that guide the model toward novel tasks. The model leverages its pre-existing knowledge to generate predictions based on the given prompt for the new task description [7]. The model is provided with no examples.

Few-shot prompting technique provides the model with a few input-output examples to induce an understanding of a given task. The selection and composition of prompt examples can significantly influence the model's behavior, leading to potential biases such as favoring frequent words, highlighting the need for caution and awareness when using prompt engineering techniques [7].

Chain-of-thought prompting facilitates a coherent and step-by-step reasoning process. It can guide the LLMs through a logical reasoning chain. The results in responses reflect a deeper understanding of the given prompts.

2.3 Applications of LLMs in the Context of Code Cbfuscation

Wu et al. [1] delve into the limits of ChatGPT by analyzing seven security applications. Their goal is to determine whether ChatGPT can comprehend the task's goal, assess the accuracy of its answers, and identify its limitations when it cannot understand the task. One security application is code decompilation, a reverse engineering technique that allows developers to understand how a program works by converting machine code back into a human-readable form. This process is similar to code deobfuscation, as the purpose is to reverse-engineer the code and make it understandable again. Code obfuscation and anti-debugging techniques [10] can significantly complicate code decompilation.

They collected LeetCode solutions written in the C programming language, which they compiled into binary code. The zero-shot prompt: "Decompile the following disassembly to C program" was then used. The responses from ChatGPT were

manually processed by extracting the decompiled source code. The evaluation used three metrics: name match, type match, and correctness. The first two compare the number of variables and functions that retain their original values, whereas correctness is evaluated by running the corresponding LeetCode test suite for each code snippet. Furthermore, they chose a well-known binary analysis tool to perform the same decompilation tasks and use its results as a baseline.

Their results show that LLMs exhibit exceptional potential in assisting with a wide range of software security tasks, demonstrating strong proficiency in processing source code and interacting with humans. With proper prompting, ChatGPT can easily understand the purpose of the tasks and generate reasonable responses. However, they have a constrained ability to process long code contexts.

In their paper, Sagodi et al. [11] investigated the functional and non-functional qualities of the code synthesized by ChatGPT and Copilot on a program synthesis benchmark containing 25 tasks. Besides assessing the quality of the LLM-generated source code, they conduct a human evaluation of the code's readability with developers with between 5 and 20 years of experience. Each of them gives a score between -2 and 2 to the LLM-generated code, considering how easy it is to follow the flow of the code, how clear the purpose of a variable or method is, and how easily the various steps and conditions can be understood.

Although LLMs are designed to interact with natural language, simple commands such as "Generate" or "Write code" can lead to the desired results. Various prompting techniques can optimize those results. The task description should challenge the model and be clear enough for the model to understand what the solution should be. The level of detail in the problem description is essential. Providing too much information does not indicate whether the model can identify complex connections between elements within the problem [11]. However, providing too little information may result in the model either not generating a solution, or the generated solutions being too different [11].

Mohseni et al. [12] analyze whether LLMs can generate new obfuscated assembly code. They use a series of baseline generative models that were either trained, zero-shot prompted, or in-context learned on the dataset they released. These models were evaluated using automatic scores and a human review to assess their obfuscation abilities.

To measure the obfuscation level, two metrics were considered. Character-wise Delta Entropy (derived from the Shannon Entropy), which is commonly used as a first-pass analysis for both original code and obfuscated code. In the context of code obfuscation, it quantifies the complexity and diversity of the code. The second measure was cosine similarity (CS) using its standard formulation to assess the similarity between the original and obfuscated code [12].

The expected delta entropy value is between 10% and 20%, which defines an effective obfuscation engine. A value exceeding this range risks altering the original code's

functionality, whereas a value below 10% indicates minimal obfuscation. The cosine similarity is expected to be above 0.9, which is essential to confirm that the initial code behavior is preserved.

In our study, we focus on the obfuscation code transformations that an LLM can apply to standalone JavaScript code, compared to an existing free-to-use obfuscator. We evaluate the results based on code correctness, a metric also used in other studies. Additionally, we use the CodeBLEU metric to assess the similarity to the original code and ultimately determine the extent of code changes resulting from LLM obfuscation. Our focus is on obfuscation code transformations rather than security-related contexts. Our approach to using the CodeBLEU as a metric to evaluate obfuscation is quite unique, as related studies we have reviewed evaluated obfuscation with the help of human expert review, or even more complex metrics, such as the one described by Mohseni et al. [12].

2.4 LLMs and Code Deobfuscation

LLMs' advanced capabilities in generating functional code, comprehending code context, and summarizing their operations can also be leveraged in reverse engineering and malware deobfuscation. While LLMs are not mature enough to fully replace traditional deobfuscators, they can efficiently complement them whenever they fail.

LLMs enable control over the randomness and creativity of their generated responses through a parameter called temperature. For their experiment, they set the temperature to zero, resulting in focused, deterministic outcomes with minimal hallucinations, thus allowing for reproducibility. The prompt they used for one of the local LLMs was: "Follow the instructions." Do not provide any explanations. Encode your responses as JSON. Your responses must start with “‘json and end with “‘." [8]

LLMs can effectively automate a substantial portion of the deobfuscation process. Local LLMs can be fine-tuned to optimize their performance in specific tasks, as their weights are publicly available. This will be explored in future work, including the investigation of smaller LLMs to provide resource-efficient solutions related to code deobfuscation and analysis. They do not foresee LLMs replacing traditional unpackers, but instead operating in existing pipelines to enrich traditional malware analysis and threat intelligence platforms [8].

2.5 Code Obfuscation Evaluation

Code obfuscation evaluation is one challenge in the field, as we have not found a general approach suitable for all obfuscation studies. In the following paragraphs, we present a few significant approaches used in related studies.

Collberg defines three obfuscation metrics in his taxonomy [6], mentioned by Rauti et al. as well [13]. These are potency, resilience, and cost. Potency refers to the degree to which a human reader is confused when they try to decipher the obfus-

cated source code and is defined as the measure of the extent to which applying obfuscation transformation T changes the complexity of program P. If the complexity of the program is increased, transformation T is said to be potent. Some applicable metrics can be program length, nesting level of conditional constructs, and cyclomatic complexity [13].

Resilience measures how well the code resists automated deobfuscation attacks, for which two metrics are needed: programmer effort and deobfuscator effort. Programmer effort represents the amount of time and space required to build an automatic deobfuscator. Measuring the effort is based on the notion that a transformation affecting a larger part of a program is more difficult to attack. Deobfuscator effort refers to the execution time and space required by the automatic deobfuscation tool to effectively reduce the potency of P.

The easy obfuscation reversibility renders high resilience and potency almost useless, because the time-consuming analysis has already been done, and the adversary has to find and use a suitable deobfuscator, which is trivial [13]. As Collberg [6] states, metrics of program obscurity, such as resilience and potency, are always quite vague by definition because they are partly based on human cognitive abilities, which makes Collberg's metrics unsuitable to be applied in this study.

Buse et al. [14] define readability as a human judgment of how easy a text is to understand. The readability of a program, however, is related to its maintainability; thus, it is a key factor in software quality. They hypothesize that programmers have some intuitive notion of readability, such as indentation, choice of identifier names, and comments. They also note that readability is not the same as complexity.

Code obfuscation can be evaluated in various ways depending on the chosen study setting. For instance, one approach is to build a labeled dataset, which helps determine whether the results returned by the LLM are correct. In this study, the experimental setting is different. There is no additional model training or labeled data to determine whether the results are successful in obfuscating or deobfuscating code [14].

In this study, we are using code correctness and the CodeBLEU metric to evaluate LLM obfuscation. Code correctness checks whether the transformed code preserves its original behavior, which is done by running a test suite and counting the number of tests that pass out of the total. Additionally, CodeBLEU helps measure how much the code has changed. CodeBLEU, as Ren et al. [15] state, is a method for automatic evaluation of code synthesis. It attempts to lower the requirement for a lexical exact match by relying on data-flow and Abstract Syntax Tree (AST) matching as well [16].

It is a valid evaluation tool because obfuscated code results can be treated as potentially valid translations of the original code. Applying the traditional BLEU directly to code synthesis will ignore the importance of the keywords, so they introduce a weighted n-gram match to assign different weights for different n-grams, so that the

keywords may have higher weights. Programming languages are manually designed and have only a few keywords [15].

In addition to the sequence-level matching, Ren et al. [15] also consider the syntactic information in CodeBLEU by matching the tree structure. Programming languages have Abstract Syntax Trees (ASTs) in which each node denotes a construct occurring in the source code. Only the syntactic structure of the code is important, and names are excluded. This score component evaluates code quality from a syntactic perspective, as grammatical errors can occur (missing tokens, data type errors), and these can be captured by the difference between their ASTs.

In programming languages, the semantics of source code are highly relevant to the dependency relations among variables. Ren et al. [15] use data flow to represent a source code as a graph, in which the nodes represent the variables and edges represent where the value of each variable comes from. Such a semantic flow graph can be used to measure the semantic match between the candidate and the reference.

3

Methods

We explore the capabilities of an LLM to obfuscate JavaScript code by conducting a laboratory experiment, as defined by Stol and Fitzgerald [17], in which we interact with the model through carefully crafted prompts, then analyze the results and compare them to those of an existing free-to-use JavaScript-compatible obfuscator. An overview of all experimental phases is shown in **Figure 3.1**.

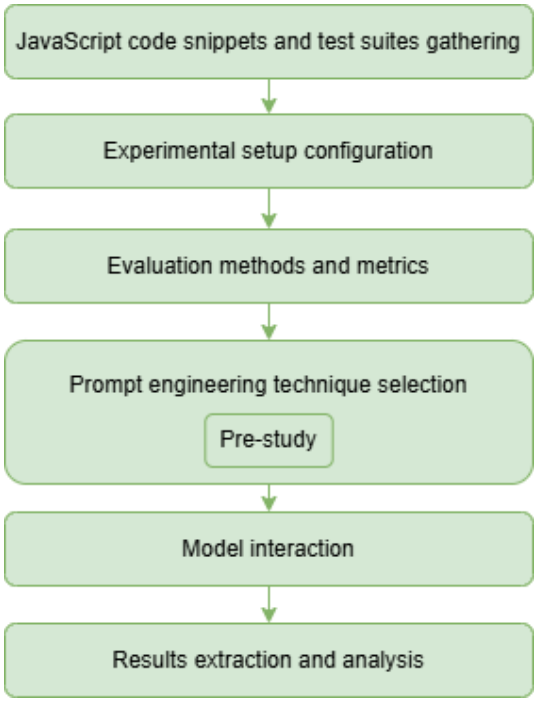

Figure 3.1: Experiment overview

The experiment begins when the dataset is compiled, consisting of JavaScript code snippets and their test suites. Then, the experimental setup is configured, comprising several key elements. The first element is the JavaScript Obfuscator tool. Due to its popularity and configurations, it is the baseline tool for evaluating our LLM-based solution. Dedicated resources using the Alvis [18] infrastructure are allocated. The isolation of the resources enabled us to install an open-weight model and interact with it through the automated pipeline we built, without any external

interaction. Alongside model interaction, it provides functionality to integrate the JavaScript obfuscator tool, result extraction and processing, test suites execution, and evaluation metrics calculations. Its purpose is to minimize human errors while efficiently automating the iterative steps of the experiments.

The next step is to establish evaluation methods and metrics that address the research questions. A prompt engineering technique is selected based on a Pre-Study we conduct, to carefully craft and apply prompts repeatedly, with ten iterations defined for each prompt configuration. The code returned by the model is then extracted and analyzed using data visualization and descriptive statistics.

In our laboratory experiment, several variables related to our research strategy can be identified. First, the controlled variable is the LLM. The prompts we build represent the independent variables, which we manipulate, while the evaluation metrics we calculate and describe later in this chapter are the dependent variables. Our dataset and isolated experimental setup emphasize precision and control.

No model parameters are changed in our study, which can be considered an element of a field experiment, as we only observe the results of various prompts. Similarly, in the interaction with JavaScript Obfuscator, we can toggle certain settings or preset configurations, without altering any internal parameters or functions. The LLM and the obfuscation tool can be considered similar to a natural setting, providing a high degree of realism.

3.1 Dataset

Similar to Wu et al. [1], the JavaScript code dataset is compiled from LeetCode [19]. The platform is easily accessible with a registered account and provides access to a wide range of coding problems and their solutions. An increased level of difficulty requires more complex elements to build the solution code, such as operators, expressions, conditionals, loops, and data types. The platform also provides test suites for each coding problem, which is essential for our study to evaluate the correctness of the code for both the original and obfuscated versions. Since these are already available on the platform, no additional effort is required to create them as part of this experiment. The test suites are executed for all code snippets in the dataset and all obfuscations performed by JavaScript Obfuscator to ensure they implement the intended behavior.

Ten coding exercises were randomly chosen for each difficulty level (easy, medium, and high), resulting in 30 code snippets and their corresponding test suites. The code corresponds to the first available JavaScript solution that other users published. The following table shows the structure of the compiled dataset for this experiment. **Appendix A** contains one code snippet corresponding to each difficulty level. **Table 3.1** shows a description of the dataset code snippets.

Difficulty level	Total number of snippets	Number of snippets defining two parameters	Number of lines of code		
			Min.	Max.	Average
Easy	10	2	3	20	13.6
Medium	10	2	17	33	23.6
High	10	6	14	64	30.1

Table 3.1: JavaScript code dataset structure

3.2 Experimental Setup

We used the open-weight **Meta-Llama-3.1-8B-Instruct** [20] for this study, by installing its **e9e39f249a16976918f6564b8830bc894c89659** snapshot on Alvis [18] through the HuggingFace [21] command line interface. One reason for choosing this model is that it is an instruction-tuned text-only model intended for assistant-like chat [20]. The second reason is the availability of resources on Alvis and the time required to perform code obfuscation in multiple iterations and across the entire code dataset, with various prompt configurations. No additional training is performed on the model before or during the experiment.

We chose a free-to-use obfuscator as the baseline for evaluating LLM code obfuscation. JavaScript Obfuscator [3], provided by Obfuscator.io [22], is a great candidate due to its popularity, extensive documentation, configuration settings, and availability as an npm package [23]. Rauti and Leppänen [13] provide a comparative study of online JavaScript obfuscators, which shows that Obfuscator.io [22] offers the largest selection of different obfuscation transformations. The tool is highly customizable. It provides four preset configurations for low, medium, high, and default obfuscation, allowing users to toggle specific settings according to their purpose. In this study, default, medium, and high preset configurations are used, as the low obfuscation preset differs from the default option only in browser-specific functionality, which is not relevant in this context.

3.3 Prompt Engineering

To establish a suitable prompt engineering technique for this experiment, we conducted a pre-study in which we crafted prompts using several common approaches. The results were manually analyzed while executing the corresponding test suites to validate the code behavior. Additionally, we analyzed the changes in structure and syntax in comparison to the original code.

3.3.1 Experimental Pre-Study

The techniques addressed in this pre-study are Zero-Shot, One-Shot, Chain-of-Thought, and Few-Shot prompting. In Zero-Shot prompting, the model must rely only on its pre-existing knowledge and ability to perform the given task. In con-

trast, One-Shot prompting provides a single example to clarify the task. Neither appeared to yield successful results, as the provided instructions did not include sufficient context on JavaScript code obfuscation.

The Chain-of-Thought technique seemed to yield promising results, with a proper set of steps to follow in code obfuscation. However, the challenge is to evaluate whether the model follows all the steps in the specified order. By providing all this information, there is not much freedom allowed for the model in applying a set of obfuscation transformations.

The last technique we evaluated during the pre-study is Few-shot prompting, which can be used to enable in-context learning, where demonstrations are provided in the prompt to steer the model to better performance. It achieved promising results in obfuscation, as seen in our initial test obfuscations. When providing the examples, the priority goal is to shape the model's understanding to create a valid and effective obfuscation. Moreover, the intention is not to mimic the existing tools, but to provide a structured launching point from which the model can generalize or innovate.

A few steps were required to refine the phrasing of the prompt using the Few-Shot approach. We chose to use three examples to mitigate the potential for lengthy text and context that the model would need to process. In the first trial, we created a unique prompt with examples belonging to distinct original code, which was then obfuscated by randomly applying a subset of settings provided by JavaScript Obfuscator. This prompt was then applied to our dataset code. The model only managed to obfuscate a few code snippets slightly, which was an indicator of the insufficient obfuscation context we provided in the examples. Updated examples did not make a significant change, which is why this approach was discarded.

The next step was to select additional code snippets from the LeetCode platform and apply several subsets of obfuscation transformations using the JavaScript Obfuscator to build three examples. Applying this prompt to our dataset revealed significant improvements; however, the results suggested that further enhancements could be achieved by incorporating difficulty levels. This led to setting up three prompt configurations, one for each difficulty level (easy, medium, and high). While calibrating the examples and running a few tests, we decided to include two additional prompt configurations to include examples with one difficulty level above. Having these in place, we proceeded further to conduct our laboratory experiment using this approach.

3.3.2 Choice of Prompt Engineering Technique

As previously described by our Experimental Pre-Study, this laboratory experiment was conducted using five configurations for Few-Shot prompting. For each difficulty level (easy, medium, high), we select an additional JavaScript solution solving a coding problem on LeetCode [19]. The additional code is distinct from any of the existing snippets in the dataset. The prompt text is generated for each code

snippet in the dataset using the examples of its corresponding difficulty level in the mapping shown in **Table 3.2**. The prompt examples are obfuscated using JavaScript Obfuscator, with various settings on, relatively random, but we also used the preset Default and Medium configurations.

Prompt Configuration	Original code snippet difficulty level	Example code snippet difficulty level
Configuration 1	Easy	Easy
Configuration 2	Easy	Medium
Configuration 3	Medium	Medium
Configuration 4	Medium	High
Configuration 5	High	High

Table 3.2: Prompt configuration mapping

Ten iterations of the prompt are executed for each configuration. The Few-Shot prompt for obfuscating code of easy difficulty using examples of the same difficulty can be found in **Appendix B**. To ensure that the model strictly relies on its existing knowledge, we construct the prompts to specifically prohibit it from referring to external tools. Without this instruction, the results include obfuscation performed by existing obfuscators, rather than the model itself. Additional training from one prompt to another is not implemented, as each obfuscation prompt interaction is independent, and the model session is cleared after every operation.

3.4 Data Analysis

Our research aims to determine the extent to which LLMs can obfuscate JavaScript code by addressing the research questions. Ideally, obfuscated code is entirely different from its original form while maintaining its intended (original) behavior. We evaluate the model results against our baseline tool, JavaScript Obfuscator. To set up the baseline obfuscations, we choose three preset setups: default, medium, and high, to which we add two custom basic setups, referred to as **Basic 1** and **Basic 2**. Both Basic 1 and Basic 2 replace the constants, variables, and numbers with hexadecimal values and function calls with a different notation. In addition, Basic 2 removes all new lines and white spaces.

3.4.1 RQ1: To what extent does the obfuscated code maintain its original behavior?

To answer this question, we evaluate the code correctness of all obfuscation results, which include both LLM and JavaScript Obfuscator results. Code correctness measures the extent to which the obfuscated code maintains the original intended behavior. To do so, we run their corresponding test suites. Each code snippet is assigned an identification number (ID) between 1 and 30, which is essential for mapping its obfuscation results with its corresponding test suite. The LLM-obfuscated

code results are also referred to as iterations. We label the obfuscation results as correct, partially correct, and incorrect depending on how many tests they pass. The correct obfuscations pass all tests, and the partially correct ones pass at least one test but have at least one test with a different status (i.e., failed or with errors). The incorrect results do not pass any tests.

3.4.2 RQ2: Is the LLM-obfuscated code significantly different from the original code?

We use the CodeBLEU [15] to address this research question. It is an automatic metric for code synthesis evaluation that measures multiple similarities and combines them into one score. To calculate it, we use the solution provided by Chernyshev [24], and our implementation workflow is shown in **Figure 3.2**.

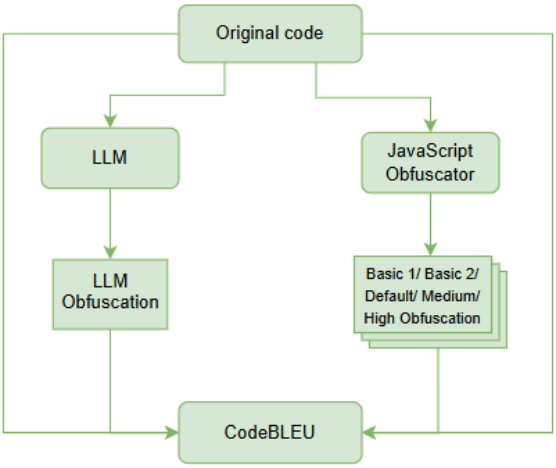

Figure 3.2: CodeBLEU metric calculation workflow

The needed arguments to calculate the value of this metric for the obfuscation results are: the reference code (the original code snippet), the candidate code (the obfuscated code), and a set of values called weights $\alpha, \beta, \gamma, \delta$, which are used to specify the importance of the metric components. All its components are relevant to determine how much the code has changed through obfuscation, and we use the default values $\alpha, \beta, \gamma, \delta = (0.25, 0.25, 0.25, 0.25)$.

First, we calculate the CodeBLEU scores of the obfuscation results performed by our baseline tool, JavaScript Obfuscator, through all five setups (Basic 1, Basic 2, Default, Medium, and High). Then, we calculate the CodeBLEU scores for all correct obfuscation iterations performed by the LLM through the five prompt configurations we defined previously (Configuration 1, Configuration 2, Configuration 3, Configuration 4, Configuration 5), also shown in **Table 3.2**. Having all these CodeBLEU scores in place allows us to evaluate the LLM obfuscation against JavaScript Obfuscator and draw experimental conclusions.

4

Results

This chapter presents the results of our laboratory experiment on code obfuscation using an LLM. By applying descriptive statistics and data visualization approaches, we present our findings and answer the research questions.

The experiment yields a total of 500 obfuscated JavaScript code snippets generated by the selected LLM. To answer RQ1, we evaluate the code correctness of all 500 results. To further evaluate the results and address RQ2, we calculate the CodeBLEU metric only for the correct iterations, which comprise 364 iterations, representing 72.8% of the initial total.

4.1 Code Correctness

4.1.1 JavaScript Obfuscator

JavaScript Obfuscator serves as our baseline for evaluating LLM obfuscations. There were five obfuscation levels set up in our pipeline for this tool: Basic 1, Basic 2, Default, Medium, and High. The tool correctly obfuscates all code snippets through each obfuscation level. Each code snippet in the dataset has its own test suite. We determined the code correctness for each snippet ID by running the test suites on all obfuscation results.

4.1.2 LLM Obfuscation

Each prompt configuration yields 100 obfuscated code snippets, 10 for each code snippet in the original dataset. One iteration or result is one LLM-obfuscated code snippet. As shown in **Table 4.1**, Configurations 1 and 2 obfuscate the first ten JavaScript code snippets in the dataset (IDs 1-10), corresponding to the easy difficulty level. Configurations 3 and 4 obfuscate the next ten (IDs 11-20), corresponding to the medium difficulty level, and Configuration 5 obfuscates the last ten (IDs 21-30), corresponding to the high difficulty level.

Table 4.2 shows the total correct, incorrect, and partially correct results for each prompt configuration. Configuration 4 yields the most correct iterations, while Configuration 1 yields the least. While Configurations 1 and 2 obfuscate the same code

Configuration	JavaScript code snippets
Configuration 1	ID 1 - ID 10
Configuration 2	ID 1 - ID 10
Configuration 3	ID 11- ID 20
Configuration 4	ID 11 - ID 20
Configuration 5	ID 21 - ID 30

Table 4.1: Overview of the JavaScript code snippets each prompt configuration obfuscates

snippets, their results differ significantly in correctness. There is a 12% improvement when more complex code examples are provided to the model in Configuration 2, which also returns 3% fewer incorrect results compared to Configuration 1. Similarly, Configuration 4 yields 16% more correct and 17% fewer incorrect iterations compared to Configuration 3. Configuration 3 and Configuration 4 return the least amount of partially correct results.

Prompt Configuration	Correct iterations (%)	Incorrect iterations (%)	Partially correct iterations (%)
Configuration 1	62	17	21
Configuration 2	74	14	12
Configuration 3	72	26	2
Configuration 4	88	9	3
Configuration 5	68	21	11

Table 4.2: Overview of correct, incorrect, and partially correct iterations for each prompt configuration

The remaining part of this section describes results for Prompt Configuration 4, which yielded the most correct obfuscation results, starting with an overview of correct and incorrect iterations of code obfuscation using an LLM. The results of the rest of the Prompt Configurations can be found in **Appendix E**. Then, we describe the errors that arose during code obfuscation with the LLM.

4.1.2.1 Prompt Configuration 4

As shown in **Table 4.2**, 9% of the total iterations are incorrect because no tests pass. Excluding these and the 88% correct iterations, a total of 3% partially correct iterations, which, for this configuration, reflect one test passing. Only ID 11, ID 13, and ID 19 have a partially correct iteration.

As shown in Figure 4.1, there are at least seven correct obfuscation results for each code snippet. Snippets ID 12, ID 15, and ID 16 are obfuscated correctly, having ten correct iterations. The rest have one incorrect iteration each, except for ID 18,

which has three.

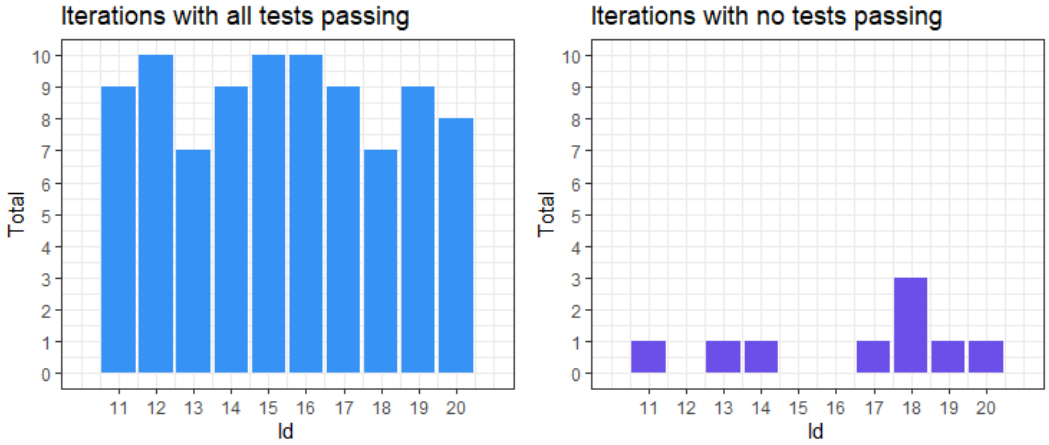

Figure 4.1: Test suites results for Prompt Configuration 4

Table 4.3 shows the incorrect, partially correct, and correct iterations for each LLM-obfuscated snippet obfuscated using Configuration 2. Its information includes the data of **Figure 4.1**, but in a more detailed form. Because the number of tests is different across the code snippets, the ones having an asterisk (∗) represent the correct iterations. Iterations passing fewer than the marked number are partially correct.

Code snippet ID	Tests in test suite	Incorrect iterations	Iterations passing 1 test	Iterations passing 2 tests	Iterations passing 3 tests
11	2	1	0	9∗	-
12	2	0	0	10∗	-
13	3	1	2	0	7∗
14	2	1	0	9∗	-
15	3	0	0	0	10∗
16	2	0	0	10∗	-
17	3	1	0	0	9∗
18	2	3	0	7∗	-
19	2	1	0	9∗	-
20	2	1	1	8∗	-

Table 4.3: Configuration 4 - Incorrect, correct, and partially correct iterations (∗ represents correct iterations)

4.1.3 Errors

Previously in this section, we have analysed the results considering passing tests. However, a test can overall have three statuses. The first one is passed, which was the focus of this chapter. However, it can also fail or have errors. A failing test means that obfuscated code has no JavaScript-specific or runtime errors, but its result was not the one expected. The third status refers to errors, and they can be specific to the JavaScript language or runtime errors. These cause the test not to execute. This section presents the errors that arose in the obfuscated code, which were detected in the code correctness process. All tests in the snippet's test suites are executed for each LLM-obfuscation iteration. Each test can either pass, fail, or have errors.

Considering all the obfuscation results, the code obfuscated by the LLM generated 110 errors, out of which 75 (68.18%) are type errors, 26 (23.64%) are reference errors, and 9 (8.18%) are range errors. **Table 4.4** shows the total errors that were thrown for each Configuration.

Configuration	Snippet IDs	Total number of errors
Configuration 1	2, 3, 5, 7, 9	19
Configuration 2	2, 4, 6, 9	22
Configuration 3	11, 12, 13, 14, 15, 17, 19	30
Configuration 4	11, 13, 17, 18	11
Configuration 5	22, 23, 24, 26, 27, 28	28

Table 4.4: Code snippet IDs that throw errors after obfuscation

The current results include three types of errors: reference, type, and range. Reference errors occur when a variable that does not exist is called somewhere in the block of code. The type error occurs when an operation could not be performed, typically (but not exclusively) when a value is not of the expected type. The range error indicates that a value is not in the set or range of allowed values. The most encountered type errors in our results are the following:

- "TypeError: Cannot read properties of undefined <element>" which means that a property of an object that has not been defined yet is accessed somewhere in the code. It can also be that functions typically applied to arrays are called on different types somewhere in the code.
- "TypeError: Assignment to constant variable". As the error says, once a constant is defined and it is assigned a value, it cannot be reassigned. In our results, this often occurs when the obfuscation assigns multiple constants at once, even if not all of them are intended to be constants.
- "RangeError: Maximum call stack size exceeded" occurs when the call stack reaches its limit, typically due to an abundance of function calls or, more

critically, infinite recursion. In our results, it occurs in recursive functions that never reach the stopping condition.

In the remaining part of this section, the most significant errors are presented for each configuration.

4.1.3.1 Prompt Configuration 1

The obfuscated code resulting from Configuration 1 yields a total of 19 errors, divided into reference and type categories, as shown in **Table 4.5**.

Error type	Percentage
Reference error	42%
Type error	58%

Table 4.5: Prompt Configuration 1 - 19 total errors

One obfuscation that results in a reference error is an iteration of ID 2. The obfuscated code can be seen in the figure below in a formatted version. The error occurs for line 5 **("eval(_0x4b9d0)['L']")** in the code below. The value of the key 'L' cannot be extracted because the **_0x4b9d0** variable is a string and has no property with the name 'L'.

```
1  var id_2 = function (c) {
2      var _0x3d2a5 = 0;
3      for (let _0x3d9100 = 0x0; _0x3d9100 < c['length']; _0x3d9100 +=
           0x1) {
4          var _0x4b9d0 = c[_0x3d9100];
5          _0x3d2a5 < eval(_0x4b9d0)['L'] ? _0x3d2a5 -= eval(_0x4b9d0)
               ['L'] : _0x3d2a5 += eval(_0x4b9d0)['L'];
6      }
7      return _0x3d2a5;
8  };
9  var _0x4b9d0 = ['I', 'V', 'X', 'L', 'C', 'D', 'M'];
10 var symbols = {
11     'I': _0x4b9d0[0x0],
12     'V': _0x4b9d0[0x1],
13     'X': _0x4b9d0[0x2],
14     'L': _0x4b9d0[0x3],
15     'C': _0x4b9d0[0x4],
16     'D': _0x4b9d0[0x5],
17     'M': _0x4b9d0[0x6]
18 };
19 module['exports'] = {'id_2': id_2};
```

One obfuscation that results in an assignment to constant variable type error is an iteration of ID 7. The following code is the formatted version of the obfuscated code. **_0x5dc20f** is a constant when defined on line 2. The error occurs on line 8, where the LLM added a reassignment of it when it obfuscated the original code.

```
var id_7 = function (_0x2d1fc5, _0x227c05) {
    const _0x2d9100 = 0x0, _0x5dc20f = _0x2d1fc5.length - 0x1;
    while (_0x2d9100 <= _0x5dc20f) {
        const _0x4e9b6d = Math.floor((_0x2d9100 + _0x5dc20f) / 0x2)
            ;
        if (_0x2d1fc5[_0x4e9b6d] == _0x227c05) {
            return _0x4e9b6d;
        } else if (_0x2d1fc5[_0x4e9b6d] > _0x227c05) {
            _0x5dc20f = _0x4e9b6d - 0x1;
        } else {
            _0x2d9100 = _0x4e9b6d + 0x1;
        }
    }
    return _0x2d9100;
};
module['exports'] = {'id_7': id_7};
```

Another obfuscation that results in a type error is an iteration of ID 9. Its error message is "TypeError: Cannot read properties of undefined (reading 'substring')". The error occurs on line 8 where a substring of the variable **e** is accessed **e.substring(g, g + 0x1)**. The issue is that **e** is firstly assigned, on line 5, to the element with index equal to 3 of the sorted original array input, which is *["flight", "flow", "flower"]*. The maximum accessible index is 2 (the length of the array minus 1). The variable **e** remains undefined, and the error is thrown.

```
var id_9 = function (b) {
    const c = '';
    b.sort();
    const d = b[0];
    const e = b[b['length'] - 0x0];
    const f = d['length'];
    for (let g = 0x0; g < f; g++) {
        if (d.substring(g, g + 0x1) == e.substring(g, g + 0x1)) {
            c += d.substring(g, g + 0x1);
        } else {
            return c;
        }
    }
    return c
};
module['exports'] = {'id_9': id_9};
```

4.1.3.2 Prompt Configuration 2

The obfuscated code resulting from Configuration 2 yields a total of 22 errors divided into reference and type categories, as shown in **Table 4.6**.

Most errors are quite similar to those in the previous configuration, including the assignment of constant variables and undefined elements. However, an iteration of ID 4 throws a different type error, which is "TypeError: Cannot mix BigInt and other types, use explicit conversions". What happened is that on line 2, an addition operation is performed between the BigInt type and *0x1*, which are not compatible.

Error type	Percentage
Reference error	32%
Type error	68%

Table 4.6: Prompt Configuration 2 - 22 total errors

The following code is the formatted version of the LLM obfuscation.

```
var id_4 = function (_0x6c4b5) {
    return (String(BigInt(_0x6c4b5.join('')) + 0x1))['split']('')['
        map'](i => Number(i));
};
module['exports'] = {'id_4': id_4};
```

4.1.3.3 Prompt Configuration 3

The obfuscated code resulting from Configuration 3 yields a total of 30 errors divided into range, reference, and type categories, as shown in **Table 4.7**.

Error type	Percentage
Range error	20%
Reference error	23%
Type error	57%

Table 4.7: Prompt Configuration 3 - 30 total errors

The reference and type errors of this configuration are quite similar to those presented earlier. However, an iteration of ID 15 throws a range error, which is "RangeError: Maximum call stack size exceeded". It occurs on line 7 on the **f.push(a[e])** instruction in the code below, which is called in a recursive function that does not reach the stop condition, and the element **a[e]** is added to the **f** array infinitely, until the stack size exceeds.

```
var id_15 = function (a, b) {
    const c = [];

    function d(e, f, g) {
        if (g === b) return c.push([...f]);
        if (g > b || e >= a['length']) return;
        f.push(a[e]);
        d(e, f, g + a[e]);
        f.pop();
        d(e + 0x1, f, g)
    }

    d(0x0, []);
    return c;
};
module.exports = {id_15};
```

4.1.3.4 Prompt Configuration 4

The obfuscated code resulting from Configuration 4 yields a total of 11 errors divided into range and type categories, as shown in **Table 4.8**.

Error type	Percentage
Range error	27%
Type error	73%

Table 4.8: Prompt Configuration 4 - 11 total errors

The errors thrown in this configuration are similar to those described in the previous configurations, as well, but an iteration of **ID 13** is returning a different type error, which is "TypeError: g(...) is not a function". The function g is defined on line 25 of the code below as a function with two parameters. The error occurs on line 22, where this function is called with only one argument. If the *"('AUxxp')"* part is removed, the code would be correct and pass all tests.

```
1  var id_13 = function (digits) {
2      if (!digits['length']) {
3          return [];
4      }
5      const digitToLetters = {
6          '2': 'abc',
7          '3': 'def',
8          '4': 'ghi',
9          '5': 'jkl',
10         '6':'mno',
11         '7': 'pqrs',
12         '8': 'tuv',
13         '9': 'wxyz'
14     };
15     const res = [];
16     function backtrack(idx, comb) {
17         if (idx === digits['length']) {
18             res['push'](comb);
19             return;
20         }
21         for (const letter of digitToLetters[digits[idx]]) {
22             backtrack(idx + 1, g('AUxxp')(comb, letter));
23         }
24     }
25     function g(k, l) {
26         return k + l;
27     }
28     backtrack(0, "");
29     return res;
30 };
31 const a = {};
32 a['id_13'] = id_13;
33 module['exports'] = a;
```

4.1.3.5 Prompt Configuration 5

The obfuscated code resulting from Configuration 5 yields a total of 28 errors divided into reference and type, as shown in the Table 4.9.

Error type	Percentage
Reference error	14%
Type error	86%

Table 4.9: Prompt Configuration 5 - 28 total errors

Similar to the previous configurations, Configuration 5 yields reference and type errors. Most of its errors refer to assignments to constant variables. However, one significant result is an iteration of **ID 28**, which throws the "TypeError: _0x2f6e3.some is not a function" error. The function "some", called on line 16 in the code below, is only applicable to arrays, but in this case, _0x2f6e3 is a number. Thus, the function cannot be applied. The correct code would be [_0x2f6e3]['some'].

```
1
2  var id_28 = function (_0x3f5b9) {
3      const _0x3e1c3 = [];
4      backtrack(_0x3e1c3, _0x3f5b9, _0x3f5b9, 0);
5      return _0x3e1c3;
6  };
7
8  function backtrack(_0x3e1c3, _0x3f5b9, _0x2f6e3 = [], _0x1d7a2 = 0)
       {
9      if (_0x1d7a2 === _0x3f5b9) {
10         _0x3e1c3['push'](_0x2f6e3['map'](function (_0x5e7f1) {
11             return '.'.repeat(_0x5e7f1) + 'Q' + '.'.repeat(_0x3f5b9
                   - _0x5e7f1 - 1);
12         }));
13         return;
14     }
15     for (let _0x4d6f1 = 0; _0x4d6f1 < _0x3f5b9; _0x4d6f1++) {
16         if (!_0x2f6e3['some'](function (_0x6f4c4, _0x4d0a3) {
17             return _0x6f4c4 === _0x4d6f1 || _0x6f4c4 === _0x4d6f1 +
                       _0x1d7a2 - _0x4d0a3 || _0x6f4c4 === _0x4d6f1 -
                   _0x1d7a2 + _0x4d0a3;
18         })) {
19             _0x2f6e3['push'](_0x4d6f1);
20             backtrack(_0x3e1c3, _0x3f5b9, _0x2f6e3, _0x1d7a2 + 1);
21             _0x2f6e3['pop']();
22         }
23     }
24 }
25
26 module.exports = { id_28 };
```

4.1.4 LLM-obfuscated Code Snippets Without Correct Iterations

The obfuscation of ID 2, using Prompt Configuration 2, yielded four iterations with no test suites passing, two partially correct iterations, and four iterations, which resulted in distinct reference and type errors, indicating that the model obfuscated this code differently.

The obfuscation of ID 27, using Prompt Configuration 5, yielded exclusively incorrect iterations. Half of the iterations resulted in a failed test suite, while the results indicate a unique error, specifically the assignment to a constant variable. The five iterations yielding this error indicate that the model wrongfully reassigns a constant variable repeatedly, which is a type error in JavaScript.

4.2 RQ 2 - CodeBLEU Evaluation

To answer the second research question, we calculated the CodeBLEU scores for all obfuscation results performed by our baseline tool and for all correct obfuscation results performed by the LLM.

The default obfuscation provided by our baseline tool serves as our primary comparison threshold, as it implements sufficient transformations to provide effective obfuscation while maintaining high code performance. Our results show that the LLM did not perform a stronger obfuscation than this threshold. Additionally, the CodeBLEU values increase with the difficulty level of the code snippets, resulting in weaker obfuscation for more complex code. In contrast, a lower CodeBLEU value indicates a stronger obfuscation. **Table 4.10** shows the CodeBLEU values (mean and standard deviation) for each prompt configuration and the corresponding baseline CodeBLEU value for the difficulty level to which the code snippets belong.

Configuration	Difficulty level	CodeBLEU LLM	CodeBLEU Baseline default
1	Easy	0.298 ± 0.056(SD)	0.245 ± 0.032(SD)
2	Easy	0.396 ± 0.095(SD)	0.245 ± 0.032(SD)
3	Medium	0.407 ± 0.107(SD)	0.183 ± 0.052(SD)
4	Medium	0.444 ± 0.122(SD)	0.183 ± 0.052(SD)
5	High	0.48 ± 0.17(SD)	0.224 ± 0.049(SD)

Table 4.10: CodeBLEU - Baseline default obfuscation and LLM results

4.2.1 JavaScript Obfuscator

Table 4.11 shows the CodeBLEU summary for each previously configured obfuscation level. The mean CodeBLEU values decrease for stronger obfuscations. Basic 1 and Basic 2 are custom levels with higher values due to the weak obfuscation we apply. However, the Default, Medium, and High obfuscations are preset setups

implemented by JavaScript Obfuscator, and their CodeBLEU values are low and do not differ significantly.

Obfuscation	Min	Max	Mean	SD	Median
Basic 1	0.139	0.563	0.315	0.082	0.315
Basic 2	0.109	0.404	0.255	0.061	0.257
Default	0.106	0.301	0.217	0.051	0.221
Medium	0.092	0.313	0.209	0.049	0.209
High	0.081	0.318	0.21	0.05	0.201

Table 4.11: CodeBLEU - Baseline results (JavaScript Obfuscator)

4.2.2 LLM Obfuscation

Out of the total 500 obfuscation iterations, 364 iterations pass all tests in their test suites, which are the candidates for CodeBLEU evaluation. We discard the iterations that alter the original code behavior. **Table 4.12** shows the summary of CodeBLEU values for each configuration of the LLM obfuscation. The complexity of the original code snippets and examples provided to the model in the prompt increases from Configuration 1 to Configuration 5. The mean values increase with this complexity, leading to weaker obfuscation for more complex code. The obfuscation is stronger if the CodeBLEU value is lower, while a higher value indicates a weaker obfuscation.

The weakest obfuscations, configured in JavaScript Obfuscator Basic 1 and Basic 2, yield maximum CodeBLEU values of 0.563 and 0.404, respectively, whereas the preset setups return values around 0.3. The LLM obfuscation returns 149 (40.93%) results with CodeBLEU > 0.4, 151 (41.48%) results with CodeBLEU in the interval of $[0.3, 0.4)$, and 64 (17.58%) results with CodeBLEU < 0.3.

Configuration	Min	Max	Mean	SD	Median
1	0.166	0.473	0.298	0.056	0.3
2	0.157	0.788	0.396	0.095	0.388
3	0.276	0.704	0.407	0.107	0.363
4	0.244	0.817	0.444	0.122	0.413
5	0.2	0.909	0.48	0.17	0.444

Table 4.12: CodeBLEU - LLM Obfuscation per obfuscation configuration

In the remainder of this section, we examine specific results of the LLM obfuscation performed on ID 9, ID 17, and ID 21, each corresponding to a different difficulty level. ID 9 has an iteration that yields the lowest CodeBLEU score, which is 0.157, while an iteration of ID 21 yields the highest score, equal to 0.909. **Table 4.13** shows the CodeBLEU results for ID 9, ID 17, and ID 21 obfuscated using JavaScript Obfuscator. In contrast, the **Table 4.14** shows the summary of the LLM obfuscations for these code snippets.

Snippet ID	Basic 1	Basic 2	Default	Medium	High
9	0.252	0.205	0.185	0.179	0.222
17	0.237	0.257	0.181	0.181	0.190
21	0.293	0.252	0.198	0.182	0.185

Table 4.13: JavaScript Obfuscator - CodeBLEU values for ID 9, ID 17, and ID 21

ID	Config	Min	Max	Mean	SD	Median
9	2	0.157	0.395	0.302	0.084	0.321
9	1	0.263	0.354	0.307	0.033	0.308
17	3	0.297	0.597	0.478	0.116	0.492
17	4	0.444	0.817	0.56	0.132	0.52
21	5	0.377	0.909	0.733	0.171	0.742

Table 4.14: LLM - CodeBLEU summary for ID 9, ID 17, and ID 21

4.2.2.1 Code snippet ID 9

The **Listing 4.1** shows the original code of ID 9 JavaScript snippet.

```
var id_9 = function (strs) {
    let prefix = ''
    strs.sort()
    const first = strs[0]
    const last = strs[strs.length - 1]
    const wordLength = first.length
    for (let i = 0; i < wordLength; i++) {
        if (first.substring(i, i + 1) === last.substring(i, i + 1))
            {
            prefix += first.substring(i, i + 1)
        }
        else { return prefix }
    }
    return prefix
};

module.exports = { id_9 };
```

Listing 4.1: Original Code - ID 9

The LLM performed the strongest obfuscation on the JavaScript snippet ID 9, with a CodeBLEU value of 0.157. The result looks somewhat similar to the Basic 2 obfuscation performed by JavaScript Obfuscator. Both obfuscations replace constants and variables with hexadecimal values, and replace explicit string function and property calls, such as ".substring()" and ".length", with bracket notation calls, like "['substring']" and "['length']". The module export statement is the same for both obfuscations. **Listing 4.2** shows the LLM obfuscation for this code snippet, performed using Configuration 2.

```
var id_9=function(_0x1d4f3){let _0x3c3d5='';_0x1d4f3['sort']();
    const _0x1f3c1=_0x1d4f3[0];const _0x4f4d=_0x1d4f3[_0x1d4f3['
    length']-0x1];const _0x5e93=_0x1f3c1['length'];for(let _0x3a7a6
    =0x0;_0x3a7a6<_0x5e93;_0x3a7a6++){if(_0x1f3c1['substring'](
    _0x3a7a6,_0x3a7a6+0x1)==_0x4f4d['substring'](_0x3a7a6,_0x3a7a6+0
    x1)){_0x3c3d5+=_0x1f3c1['substring'](_0x3a7a6,_0x3a7a6+0x1);}
    else{return _0x3c3d5;}}return _0x3c3d5;};module['exports']={'
    id_9':id_9};
```

Listing 4.2: LLM Obfuscation - ID 9, Configuration 2, CodeBLEU = 0.157

The next obfuscation iteration of ID 9, shown in Listing 4.3, was performed using the Prompt Configuration 1 and yielded a CodeBLEU score of 0.263. Comparing this result with the JavaScript obfuscator, it seems similar to the result of Basic 2 obfuscation. The constants and variables are replaced with hexadecimal values; the module export statement is similar as well. One difference is that the LLM calls the ".sort()" and ".substring()" functions directly, rather than replacing them with the bracket notation seen earlier.

```
var id_9=function(_0x3a5e1){let _0x2e0f9='';_0x3a5e1.sort();const
    _0x53a3=_0x3a5e1[0];const _0x4e79=_0x3a5e1[_0x3a5e1['length']-0
    x1];const _0x5a7b=_0x53a3['length'];for(let _0x4d5f=0x0;_0x4d5f<
    _0x5a7b;_0x4d5f++){if(_0x53a3.substring(_0x4d5f,_0x4d5f+0x1)==
    _0x4e79.substring(_0x4d5f,_0x4d5f+0x1)){_0x2e0f9+=_0x53a3.
    substring(_0x4d5f,_0x4d5f+0x1);}else{return _0x2e0f9;}}return
    _0x2e0f9;};module['exports']={'id_9':id_9};
```

Listing 4.3: LLM Obfuscation - ID 9, Configuration 1, CodeBLEU = 0.263

The LLM obfuscation of ID 9 shown in Listing 4.4 was performed using Configuration 1 and yielded a CodeBLEU value of 0.313. Its code is very similar to the Basic 2 obfuscation of JavaScript Obfuscator. The only noticeable difference is that the LLM replaced the original variable and constant names with alphabetical English letters and not hexadecimal values. This transformation is called name mangle.

```
var id_9=function(b){let a='';b.sort();const c=b[0];const d=b[b.
    length-1];const e=c.length;for(let f=0x0;f<e;f++){if(c.substring
    (f,f+0x1)===d.substring(f,f+0x1)){a+=c.substring(f,f+0x1);}else{
    return a;}}return a;};module['exports']={'id_9':id_9};
```

Listing 4.4: LLM Obfuscation - ID 9, Configuration 1, CodeBLEU = 0.313

The following obfuscation of ID 9 is also performed by the LLM using Prompt Configuration 2, having a CodeBLEU value of 0.395, and is presented in Listing 4.5. This result is somewhat similar to the Basic 1 obfuscation done with our baseline tool, JavaScript Ofuscator. The constants and variables seem to be mangled by the LLM, which means that they are replaced with alphabetical English lowercase letters, in contrast with the baseline tool. However, the string-specific function ".substring()" and the property ".length" are replaced with bracket notation in both obfuscations.

```
var id_9 = function (a) {
    let b = '';
```

```
    a['sort']();
    const c = a[0];
    const d = a[a['length'] - 0x1];
    const e = c['length'];
    for (let f = 0; f < e; f++) {
        if (c['substring'](f, f + 0x1) === d['substring'](f, f + 0
            x1)) {
            b += c['substring'](f, f + 0x1);
        }
        else { return b }
    }
    return b
};
module.exports = { id_9 };
```

Listing 4.5: LLM Obfuscation - ID 9, Configuration 2, CodeBLEU = 0.395

The obfuscations performed by the preset options of JavaScript Obfuscator are significantly more advanced and challenging to understand. The default obfuscation of ID 9 using the JavaScript Obfuscator can be seen below in **Listing 4.6**, **Listing 4.7**, and **Listing 4.8**. In contrast, those corresponding to medium and high obfuscations can be observed in the **Appendix E**.

```
var id_9 = function (_0x4169d0) {
    let _0x27939e = '';
    _0x4169d0['sort']();
    const _0x6104d5 = _0x4169d0[0x0];
    const _0x5a6c04 = _0x4169d0[_0x4169d0['length'] - 0x1];
    const _0x45e07c = _0x6104d5['length'];
    for (let _0x386c60 = 0x0; _0x386c60 < _0x45e07c; _0x386c60++) {
        if (_0x6104d5['substring'](_0x386c60, _0x386c60 + 0x1) ===
            _0x5a6c04['substring'](_0x386c60, _0x386c60 + 0x1)) {
            _0x27939e += _0x6104d5['substring'](_0x386c60,
                _0x386c60 + 0x1);
        } else {
            return _0x27939e;
        }
    }
    return _0x27939e;
};
module['exports'] = { 'id_9': id_9 };
```

Listing 4.6: JavaScript Obfuscator - ID 9, Basic 1, CodeBLEU = 0.252

```
var id_9=function(_0x265dcd){let _0x27d088='';_0x265dcd['sort']();
    const _0x457e0e=_0x265dcd[0x0];const _0x43d3b2=_0x265dcd[
    _0x265dcd['length']-0x1];const _0x475f0b=_0x457e0e['length'];for
    (let _0xd8c94c=0x0;_0xd8c94c<_0x475f0b;_0xd8c94c++){if(_0x457e0e
    ['substring'](_0xd8c94c,_0xd8c94c+0x1)===_0x43d3b2['substring'](
    _0xd8c94c,_0xd8c94c+0x1)){_0x27d088+=_0x457e0e['substring'](
    _0xd8c94c,_0xd8c94c+0x1);}else{return _0x27d088;}}return
    _0x27d088;};module['exports']={'id_9':id_9};
```

Listing 4.7: JavaScript Obfuscator - ID 9, Basic 2, CodeBLEU = 0.205

```
const _0x2036f9=_0x5556;(function(_0x1104bf,_0x4369d7){const
    _0x3d062a=_0x5556,_0x508a1e=_0x1104bf();while(!![]){try{const
    _0x182a81=-parseInt(_0x3d062a(0x1dc))/0x1*(-parseInt(_0x3d062a(0
    x1d8))/0x2)+-parseInt(_0x3d062a(0x1db))/0x3*(-parseInt(_0x3d062a
    (0x1e0))/0x4)+-parseInt(_0x3d062a(0x1da))/0x5*(-parseInt(
    _0x3d062a(0x1d2))/0x6)+parseInt(_0x3d062a(0x1d4))/0x7*(-parseInt
    (_0x3d062a(0x1d0))/0x8)+-parseInt(_0x3d062a(0x1d6))/0x9*(-
    parseInt(_0x3d062a(0x1d1))/0xa)+parseInt(_0x3d062a(0x1d7))/0xb+
    parseInt(_0x3d062a(0x1d9))/0xc*(-parseInt(_0x3d062a(0x1d5))/0xd)
    ;if(_0x182a81===_0x4369d7)break;else _0x508a1e['push'](_0x508a1e
    ['shift']());}catch(_0x477d2e){_0x508a1e['push'](_0x508a1e['
    shift']());}}}(_0x41c1,0xd1d6b));function _0x41c1(){const
    _0x320a21=['5059465HioXPI','92436IxVHMx','4EBjegB','exports','
    sort','length','208SgXnDf','762952zSkcMH','60gNuDGT','6sURByp','
    substring','7zWRZoI','338jZLLqY','1897020mKYccJ','16740867CuKEcm
    ','451538qtSJEN','2468724hkOIVi'];_0x41c1=function(){return
    _0x320a21;};return _0x41c1();}function _0x5556(_0x2c9a04,
    _0x54a8e5){const _0x41c154=_0x41c1();return _0x5556=function(
    _0x55563c,_0x5cf950){_0x55563c=_0x55563c-0x1d0;let _0x49caf7=
    _0x41c154[_0x55563c];return _0x49caf7;},_0x5556(_0x2c9a04,
    _0x54a8e5);}var id_9=function(_0x54a9d9){const _0x2c8bf1=_0x5556
    ;let _0x12db11='';_0x54a9d9[_0x2c8bf1(0x1de)]();const _0x4b7ac0=
    _0x54a9d9[0x0],_0x26a0a9=_0x54a9d9[_0x54a9d9[_0x2c8bf1(0x1df)]-0
    x1],_0x2181a2=_0x4b7ac0[_0x2c8bf1(0x1df)];for(let _0x491da2=0x0;
    _0x491da2<_0x2181a2;_0x491da2++){if(_0x4b7ac0[_0x2c8bf1(0x1d3)](
    _0x491da2,_0x491da2+0x1)===_0x26a0a9['substring'](_0x491da2,
    _0x491da2+0x1))_0x12db11+=_0x4b7ac0[_0x2c8bf1(0x1d3)](_0x491da2,
    _0x491da2+0x1);else return _0x12db11;}return _0x12db11;};module[
    _0x2036f9(0x1dd)]={'id_9':id_9};
```

Listing 4.8: JavaScript Obfuscator - ID 9, Default, CodeBLEU = 0.185

4.2.2.2 Code snippet ID 17

The original code of JavaScript snippet ID 17 is shown in Listing 4.9. The following two iterations of ID 17 are pretty similar to the Basic 2 obfuscation result of JavaScript Obfuscator, which is shown in Listing 4.14. The rest of the obfuscations corresponding to ID 17 performed with this tool can be found in **Appendix E**.

```
var id_17 = function(x, n) {
    function calc_power(x, n) {
        if (x === 0) {
            return 0;
        }
        if (n === 0) {
            return 1;
        }

        let res = calc_power(x, Math.floor(n / 2));
        res = res * res;

        if (n % 2 === 1) {
            return parseFloat(res * x).toFixed(5);
        }

```

```
17          return parseFloat(res).toFixed(5);
18      }
19
20      let ans = calc_power(x, Math.abs(n));
21
22      if (n >= 0) {
23          return parseFloat(ans).toFixed(5);
24      }
25
26      return parseFloat(1 / ans).toFixed(5);
27  }
28
29  module.exports = { id_17 };
```

Listing 4.9: Original code - ID 17

One obfuscation of ID 17, which has quite a low CodeBLEU value of 0.297, indicating a stronger obfuscation, can be found in **Listing 4.10**. It was obtained using Prompt Configuration 3. The variables and the inner function are mangled, meaning that they are replaced with English alphabetical letters. All spaces have been removed, and the export module explicitly exposes the"id_17" name, but as a key-value pair object.

```
1  var id_17=function(a,b){function c(d,e){if(d===0)return 0;if(e===0)
       return 1;let f=c(d,Math.floor(e/2));f=f*f;if(e%2===1)return
       parseFloat(f*a).toFixed(5);return parseFloat(f).toFixed(5);}let
       g=c(a,Math.abs(b));if(b>=0)return parseFloat(g).toFixed(5);
       return parseFloat(1/g).toFixed(5);};module.exports={'id_17':
       id_17};
```

Listing 4.10: LLM Obfuscation - ID 17, Configuration 3, CodeBLEU = 0.297

Another obfuscation iteration of this code snippet yielded a CodeBLEU value of 0.444, and it is shown in Listing 4.11. It was obtained using Prompt Configuration 4. The code is written on a single line, and unnecessary spaces have been removed. The names of constants and variables have not been changed, except for the module export object, which includes a hexadecimal value.

```
1  var id_17 = function(x,n){function calc_power(x,n){if(x===0)return
       0;if(n===0)return 1;let res=calc_power(x,Math.floor(n/2));res=
       res*res;if(n%2===1)return parseFloat(res*x).toFixed(5);return
       parseFloat(res).toFixed(5);}let ans=calc_power(x,Math.abs(n));if
       (n>=0)return parseFloat(ans).toFixed(5);return parseFloat(1/ans)
       .toFixed(5);};const _0x3f7e3={};_0x3f7e3['id_17']=id_17;module['
       exports']=_0x3f7e3;
```

Listing 4.11: LLM Obfuscation - ID 17, Configuration 4, CodeBLEU = 0.444

The following obfuscation was obtained through Prompt Configuration 3 and yielded a CodeBLEU value of 0.554. Listing 4.12 shows the code of this result. The variables and some numbers are replaced with hexadecimal values. However, the spaces and new lines have not been removed, and the module export statement is the same as in the original code.

```
var id_17 = function(_0x17, _0x6d) {
    function _0x1f0(_0x13, _0x11) {
        if (_0x13 === 0) {
            return 0;
        }
        if (_0x11 === 0) {
            return 1;
        }
        let _0x2f = _0x1f0(_0x13, Math.floor(_0x11 / 2));
        _0x2f = _0x2f * _0x2f;
        if (_0x11 % 2 === 1) {
            return parseFloat(_0x2f * _0x13).toFixed(5);
        }
        return parseFloat(_0x2f).toFixed(5);
    }
    let _0x5 = _0x1f0(_0x17, Math.abs(_0x6d));
    if (_0x6d >= 0) {
        return parseFloat(_0x5).toFixed(5);
    }
    return parseFloat(1 / _0x5).toFixed(5);
}
module.exports = { id_17 };
```

Listing 4.12: LLM Obfuscation - ID 17, Configuration 3, CodeBLEU 0.554

The last obfuscation result for ID 17, which we present, was obtained by applying Prompt Configuration 4 and yielded a CodeBLEU value of 0.817. This iteration is quite similar to the original code, as the variables, function names, and numbers have not been altered during code obfuscation. However, the export module is encapsulated in a hexadecimal named object containing a key-value pair object, with the key equal to the snippet ID 17. The Math functions are also used as object properties written in bracket notation, "Math['floor']" and "Math['abs']", instead of directly using their names.

```

var id_17 = function(x, n) {
    function calc_power(x, n) {
        if (x === 0) {
            return 0;
        }
        if (n === 0) {
            return 1;
        }
        let res = calc_power(x, Math['floor'](n / 2));
        res = res * res;
        if (n % 2 === 1) {
            return parseFloat(res * x).toFixed(5);
        }
        return parseFloat(res).toFixed(5);
    }
    let ans = calc_power(x, Math['abs'](n));
    if (n >= 0) {
        return parseFloat(ans).toFixed(5);
    }
    return parseFloat(1 / ans).toFixed(5);
```

```
};
const _0x5f3a3 = {};
_0x5f3a3['id_17'] = id_17;
module['exports'] = _0x5f3a3;
```

Listing 4.13: LLM obfuscation - ID 17, Configuration 4, CodeBLEU = 0.817

```
var id_17=function(_0x315f35,_0x9f7a6e){function _0x1dcd28(
    _0xcf66ad,_0x4870cf){if(_0xcf66ad===0x0){return 0x0;}if(
    _0x4870cf===0x0){return 0x1;}let _0x4ad5ab=_0x1dcd28(_0xcf66ad,
    Math['floor'](_0x4870cf/0x2));_0x4ad5ab=_0x4ad5ab*_0x4ad5ab;if(
    _0x4870cf%0x2===0x1){return parseFloat(_0x4ad5ab*_0xcf66ad)['
    toFixed'](0x5);}return parseFloat(_0x4ad5ab)['toFixed'](0x5);}
    let _0x3eafab=_0x1dcd28(_0x315f35,Math['abs'](_0x9f7a6e));if(
    _0x9f7a6e>=0x0){return parseFloat(_0x3eafab)['toFixed'](0x5);}
    return parseFloat(0x1/_0x3eafab)['toFixed'](0x5);};module['
    exports']={'id_17':id_17};
```

Listing 4.14: JavaScript Obfuscator - ID 17, Basic 2, CodeBLEU = 0.257

4.2.2.3 Code snippet ID 21

The original JavaScript code of snippet ID 21 is shown in Listing 4.15. One LLM obfuscation iteration of this snippet yielded the highest CodeBLEU score of 0.909. However, the obfuscation results of this code snippet are presented in the following paragraphs in ascending order by their CodeBLEU values. All obfuscation results for this code snippet are obtained by applying Prompt Configuration 5. These iterations seem somewhat similar to the Basic 1 obfuscation performed by the JavaScript Obfuscator, which is shown in Listing 4.20. The custom Basic 2 obfuscation and the preset setups (Default, Medium, and High) performed by JavaScript Obfuscator are provided in **Appendix E**.

```
var id_21 = function (s, p) {
    const dp = Array.from({ length: s.length + 1 }, () => Array.
       from({ length: p.length + 1 }));
    dp[s.length][p.length] = true;

    const dfs = (i, j) => {
        if (dp[i][j] !== undefined) return dp[i][j];
        const match = i < s.length && (p[j] === '.' || s[i] === p[j
           ]);
        if (j + 1 < p.length && p[j + 1] === '*') {
            dp[i][j] = dfs(i, j + 2) || match && dfs(i + 1, j);
        } else {
            dp[i][j] = match && dfs(i + 1, j + 1);
        }
        return dp[i][j];
    }
    return dfs(0, 0);
};

module.exports = { id_21 };
```

Listing 4.15: Original code - ID 21

One LLM obfuscation of snippet ID 21 shown in **Listing 4.16** yielded a CodeBLEU value of 0.377. The LLM has not removed empty lines and spaces. Overall, the obfuscation appears to be quite strong. The constants and variables are replaced with hexadecimal values. The bracket notation is used for the "length" property. The inner function, initially defined on line 5, became an anonymous arrow function.

```
1  var id_21 = function(_0x3a6c9, _0x1f4a4) {
2      const _0x4f4d = Array.from({ length: _0x3a6c9['length'] + 1 },
           () => Array.from({ length: _0x1f4a4['length'] + 1 }));
3      _0x4f4d[_0x3a6c9['length']][_0x1f4a4['length']] = true;
4      const _0x5c7a6 = (_0x2a2d, _0x5a7b) => {
5          if (_0x4f4d[_0x2a2d][_0x5a7b]!== undefined) return _0x4f4d[
               _0x2a2d][_0x5a7b];
6          const _0x5a92 = _0x2a2d < _0x3a6c9['length'] && (_0x1f4a4[
               _0x5a7b] === '.' || _0x3a6c9[_0x2a2d] === _0x1f4a4[
               _0x5a7b]);
7          if (_0x5a7b + 1 < _0x1f4a4['length'] && _0x1f4a4[_0x5a7b +
               1] === '*') {
8              _0x4f4d[_0x2a2d][_0x5a7b] = _0x5c7a6(_0x2a2d, _0x5a7b +
                   2) || _0x5a92 && _0x5c7a6(_0x2a2d + 1, _0x5a7b);
9          } else {
10             _0x4f4d[_0x2a2d][_0x5a7b] = _0x5a92 && _0x5c7a6(_0x2a2d
                   + 1, _0x5a7b + 1);
11         }
12         return _0x4f4d[_0x2a2d][_0x5a7b];
13     };
14     return _0x5c7a6(0, 0);
15 };
16 module.exports = { id_21 };
```

Listing 4.16: LLM Obfuscation - ID 21, Configuration 5, CodeBLEU = 0.377

The next iteration of ID 21 is shown in **Listing 4.17** and yielded a CodeBLEU value of 0.530. This code is quite similar to the previous obfuscation. However, the "length" property is accessed directly without using the bracket notation. The constants and variables are also replaced with hexadecimal values.

```
1  var id_21 = function (_0x11e0, _0x2e1f) {
2      const _0x1a1f = Array.from({
3          length: _0x11e0.length + 1
4      }, () => Array.from({
5          length: _0x2e1f.length + 1
6      }));
7      _0x1a1f[_0x11e0.length][_0x2e1f.length] = true;
8      const _0x3f5b = (_0x2f8a, _0x5e15) => {
9          if (_0x1a1f[_0x2f8a][_0x5e15]!== undefined) return _0x1a1f[
               _0x2f8a][_0x5e15];
10         const _0x4a0f = _0x2f8a < _0x11e0.length && (_0x2e1f[
               _0x5e15] === '.' || _0x11e0[_0x2f8a] === _0x2e1f[_0x5e15
               ]);
11         if (_0x5e15 + 1 < _0x2e1f.length && _0x2e1f[_0x5e15 + 1]
               === '*') {
12             _0x1a1f[_0x2f8a][_0x5e15] = _0x3f5b(_0x2f8a, _0x5e15 +
                   2) || _0x4a0f && _0x3f5b(_0x2f8a + 1, _0x5e15);
13         } else {
```

```
14              _0x1a1f[_0x2f8a][_0x5e15] = _0x4a0f && _0x3f5b(_0x2f8a
                    + 1, _0x5e15 + 1);
15          }
16          return _0x1a1f[_0x2f8a][_0x5e15];
17      };
18      return _0x3f5b(0, 0);
19  };
20  module.exports = { id_21 };
```

Listing 4.17: LLM Obfuscation - ID 21, Configuration 5, CodeBLEU = 0.530

The following obfuscation of ID 21, shown in Listing 4.18, is slightly weaker than the previous ones, having a CodeBLEU score of 0.702. The obfuscated code appears to be very similar to the original code. Only a few constants (including the function on line 4) were replaced with hexadecimal values. The module export approach also differs from the original code. Instead of directly exporting the function with its original name, an object that contains it was exported instead.

```
1   var id_21 = function (s, p) {
2       const dp = Array.from({length: s.length + 1}, () => Array.from
            ({length: p.length + 1}));
3       dp[s.length][p.length] = true;
4       const _0x1a1a = (i, j) => {
5           if (dp[i][j]!== undefined) return dp[i][j];
6           const _0x1f3d = i < s.length && (p[j] === '.' || s[i] === p
                [j]);
7           if (j + 1 < p.length && p[j + 1] === '*') {
8               dp[i][j] = _0x1a1a(i, j + 2) || _0x1f3d && _0x1a1a(i +
                    1, j);
9           } else {
10              dp[i][j] = _0x1f3d && _0x1a1a(i + 1, j + 1);
11          }
12          return dp[i][j];
13      }
14      return _0x1a1a(0, 0);
15  };
16  const _0x3cbb = {};
17  _0x3cbb['id_21'] = id_21;
18  module['exports'] = _0x3cbb;
```

Listing 4.18: LLM Obfuscation - ID 21, Configuration 5, CodeBLEU = 0.702

The last obfuscation of ID 21 we present is much more similar to the original code than the previous one. It is shown in Listing 4.19. There are two noticeable changes, though. The first one is that the arrow function on line 4 is written as a standard function, and the second is the module export, which is similar to the obfuscations presented previously. The rest of the obfuscated code is identical to the original form.

```
1   var id_21 = function (s, p) {
2       const dp = Array.from({ length: s.length + 1 }, () => Array.
            from({ length: p.length + 1 }));
3       dp[s.length][p.length] = true;
4       const dfs = function (i, j) {
```

```
5         if (dp[i][j]!== undefined) return dp[i][j];
6         const match = i < s.length && (p[j] === '.' || s[i] === p[j
              ]);
7         if (j + 1 < p.length && p[j + 1] === '*') {
8             dp[i][j] = dfs(i, j + 2) || match && dfs(i + 1, j);
9         } else {
10            dp[i][j] = match && dfs(i + 1, j + 1);
11        }
12        return dp[i][j];
13    }
14    return dfs(0, 0);
15};
16const _0x5f1a3 = {};
17_0x5f1a3['id_21'] = id_21;
18module['exports'] = _0x5f1a3;
```

Listing 4.19: LLM Obfuscation - ID 21, Configuration 5, CodeBLEU = 0.909

```
1 var id_21 = function (_0x448eba, _0x3e1814) {
2     const _0x3524b9 = Array['from']({ 'length': _0x448eba['length']
          + 0x1 }, () => Array['from']({ 'length': _0x3e1814['length'
          ] + 0x1 }));
3     _0x3524b9[_0x448eba['length']][_0x3e1814['length']] = !![];
4     const _0x163d2c = (_0x48a048, _0x2554a0) => {
5         if (_0x3524b9[_0x48a048][_0x2554a0] !== undefined)
6             return _0x3524b9[_0x48a048][_0x2554a0];
7         const _0x3887a4 = _0x48a048 < _0x448eba['length'] && (
              _0x3e1814[_0x2554a0] === '.' || _0x448eba[_0x48a048] ===
              _0x3e1814[_0x2554a0]);
8         if (_0x2554a0 + 0x1 < _0x3e1814['length'] && _0x3e1814[
              _0x2554a0 + 0x1] === '*') {
9             _0x3524b9[_0x48a048][_0x2554a0] = _0x163d2c(_0x48a048,
                  _0x2554a0 + 0x2) || _0x3887a4 && _0x163d2c(_0x48a048
                  + 0x1, _0x2554a0);
10        } else {
11            _0x3524b9[_0x48a048][_0x2554a0] = _0x3887a4 &&
                  _0x163d2c(_0x48a048 + 0x1, _0x2554a0 + 0x1);
12        }
13        return _0x3524b9[_0x48a048][_0x2554a0];
14    };
15    return _0x163d2c(0x0, 0x0);
16};
17module['exports'] = { 'id_21': id_21 };
```

Listing 4.20: JavaScript Obfuscator - ID 21, Basic 1, CodeBLEU = 0.293

5

Discussion

5.1 Research Question 1

Research Question 1 asks to what extent the LLM-obfuscated code maintains its original behavior. The results indicate that LLMs can generally obfuscate code. However, the results can be unpredictable. The choice of prompt engineering technique, structure, and phrasing can make a significant difference. It is essential to note that, unless specifically instructed not to, the model utilizes external tools and resources to obfuscate the provided code. The results seem successful in these cases, as the model does not do the obfuscation itself.

Code obfuscation can be more challenging than other code synthesis tasks. Code can be obfuscated in many ways, and a block of code of a few lines can easily become a block of several hundred lines of code. There are also several obfuscation transformations applied in a sequence in the existing solutions. It is not easy to fully control how the LLM obfuscates code. Building an optimal prompt that facilitates obfuscation of different complexities, involving more than variable renaming and rewriting the code on one line, is a challenge.

In our pre-study, we tried several approaches before selecting the few-shot prompt engineering technique. We first manually checked if the resulting code differed from the original input, and then we ran the test suites to determine whether the behavior had changed. Having this module of the pipeline helped determine the optimal prompt engineering technique to use for this study.

The Few-Shot technique, with customized examples for each difficulty level of the dataset, returned promising results. When we decided to use Few-Shot prompting, the initial tests included providing examples of one specific difficulty level. Then, we compiled a set of three examples of different difficulty levels. Both options performed correctly on only a few code snippets.

Considering advanced obfuscation, which results in hundreds of lines of code, is another challenge in building prompts. We were unable to include overly large obfuscation examples due to errors encountered during the model interaction, which impacted the overall correctness of Configuration 5 described in the previous chapter.

The Chain-of-Thought seemed to be a promising approach to improve and build a more standardized and predictable obfuscation. However, this limits the LLM's capability to perform the task, as the obfuscation steps would be defined for the model to follow strictly, like how existing tools implement specific algorithms.

Overall, considering code correctness, the baseline tool, JavaScript Obfuscator, performed an obfuscation significantly better than our LLM-based solution. The code obfuscation by this tool maintained the original behavior in all obfuscations.

5.2 Research Question 2

Research Question 2 asks if the LLM-obfuscated code is significantly different than the original code. Ideally, when obfuscating code, the result should be as different as possible from the original code and difficult to understand and follow the code flow. As we conducted a laboratory experiment, our goal was to find a metric to help us evaluate code obfuscation.

The metric we chose, CodeBLEU, is a code synthesis evaluation metric that proved suitable for our study. By taking multiple aspects into account and calculating an overall score based on those, it provides a way to assess how much the code has changed through obfuscation compared to the original. It not only evaluates the n-gram match, but it includes a custom weighted n-gram match specifically tailored for programming languages that have a limited set of keywords. Its syntax match component compares the abstract syntax trees of the original code and the generated code, while the data-flow match component maps the relations among the variables.

Initially, besides code correctness, our second evaluation method was to assess the overall readability of the obfuscated code, which proved more challenging than anticipated, as we were unable to find any related work that evaluates code obfuscation from this perspective without incorporating human expert review. Code readability is a metric of maintainability in software. However, considering obfuscation, the goal is to obtain an entirely different code from which it is difficult to extract implementation details. We then attempted to explain readability by evaluating its similarities to the original code using the CodeBLEU metric. These similarities determine how much the code has been altered during the obfuscation process.

The CodeBLEU values turned out to be as we initially expected. Stronger obfuscations yield lower scores, while weaker obfuscations yield higher values. The LLM proved capable of applying basic obfuscation transformations, but the prompt engineering technique and phrasing have a significant impact on the result. The prompts we used can be further improved, and perhaps combining more techniques could yield even better results.

Our first assumption was that a strong obfuscation should yield a low CodeBLEU score. To verify this, we analyzed the tool's obfuscation on our dataset snippets at the five levels of obfuscation we configured – Basic 1, Basic 2, Default, Medium, and High, where Basic 1 provides the weakest obfuscation and High provides the

strongest. The CodeBLEU values for all these levels of obfuscation showed that our intuition was correct, as the values decreased considerably from Basic 1 to Default, while the values between Default and High are quite similar but slightly decreasing. This is because the JavaScript Obfuscator already provides a strong Default obfuscation, which is optimized to maintain high code performance. Medium and High levels provide stronger obfuscation, but can significantly impact code performance. Table 4.11 shows the mean values and the standard deviation for each of these obfuscation levels. The mean and standard deviation are calculated for all obfuscated results per level (snippets 1-30).

The existing JavaScript obfuscation tools implement a set of transformations that are always applied to the code in the same order, depending on the default or selected configurations. However, the purpose of our study was not to evaluate whether the LLM can follow specific steps and instructions to obfuscate code, but to determine if it is possible to extract obfuscation patterns already applied to the code. In most cases, the LLM extracted at least one obfuscation pattern from the examples we provided in the prompt, and some results are very similar to those obtained by configuring the basic obfuscation levels using JavaScript Obfuscator. Manually assessing the code of these results, we could determine a few of the code transformations applied by the LLM, such as formatting or variable and property renaming. However, the baseline tool, JavaScript Obfuscator, yielded stronger obfuscations compared with our LLM-based solution. Some of the more complex code snippets resulted in more accurate results compared to the least complex code snippets (easy difficulty level). Code correctness is not sufficient to conclude why this is the case. However, the increased CodeBLEU values indicate that the code has not changed significantly, suggesting that the obfuscation results are less prone to contain errors.

5.3 Threats to Validity

5.3.1 Internal Validity

The first internal threat to validity we identified is the JavaScript dataset. In real-world scenarios, obfuscation is applied to JavaScript code embedded in HTML pages. This code can contain simple instructions similar to our dataset code snippet, but it can also include functionality to manipulate the Document Object Model (DOM) elements of the HTML pages. This includes, for instance, the document object, which is the owner of all other objects in the given web page and provides functionality to select specific elements, change their content, and add or remove elements. Even if our code snippets are written in JavaScript, they do not cover functionality often integrated in web or plain HTML pages.

We have independent code snippets. More complicated to include dependencies on other code snippets or third-party modules. Validating whether the code maintains the initial behavior would be much more difficult, as it would require a more advanced pipeline. However, even if this is a more realistic setup, independent code

snippets are still necessary to study the code obfuscation transformations at this stage, rather than evaluate whether the model can successfully obfuscate dependencies.

We gathered the JavaScript code snippets and their test suites from a coding preparation platform. As most of them have between two and three tests in their suites, some functionality exceptions or corner cases may not be included. Even if the obfuscated code passes all tests, it is not guaranteed that it maintains the intended initial behavior when corner cases are encountered.

The second internal threat to validity is our selection of LLM. We used a relatively small model, and the results cannot be generalized to other LLMs, as they are specific to our experimental setup, the selected LLM, and the prompt engineering technique chosen to interact with it. We conducted our laboratory experiment with only one and a relatively small LLM, first because we did not know what results to expect, and second, due to the limited available resources. The fact that the chosen LLM can obfuscate code on its own with carefully crafted prompts opens the possibility of future studies involving more advanced models.

The third internal threat to validity we identified is our prompt engineering strategy. The examples we provide to the LLM do not steer the results towards any particular obfuscation techniques. Our approach was to build obfuscated examples that contain a few simple transformations, so that the LLM could extract them and then apply them further to the input code. Additionally, due to time constraints, we were unable to conduct a thorough Pre-Study to examine more prompts with additional examples and provide a few instructions to the model as well. The Pre-Study we conducted ended when we observed an improvement in the results, indicating that the LLM can apply obfuscation transformations.

Explicitly shaping the target objectives of low readability and high correctness was not the focus of our study, as we wanted to limit the context strictly to obfuscation transformations. It could be a great idea to explore ways to improve our results, however, we are uncertain about their effectiveness. There is a possibility that these objectives could cause the model to hallucinate and alter the code in a completely different direction, rather than applying obfuscation, and instead change it in a different direction solely to achieve the goal while disregarding the examples and the focus on obfuscation.

We consider that there is a potential alignment between the types of examples we select for Few-Shot prompting and the strategies the LLM tends to apply. However, with an increased level of obfuscation reflected by the examples, the model did not seem to perform well. Examining all the prompt configurations we established, the CodeBLEU values tend to increase, particularly in Configurations 4 and 5. The increased CodeBLEU values indicate that the code has not changed significantly, and that the examples may not have been suitable for the model to extract the already existing code transformations and apply them further to our dataset code.

In our Pre-Study, we observed that applying obfuscated versions of distinct code, however, did not improve the results, as the model could not extract many transformations, which is why this approach was discarded. We also attempted to implement examples that apply a single code transformation. The main issue is that the transformation may not be applicable to all code snippets in our dataset, thereby limiting the possible obfuscation transformations. There are general transformations that can be applied, such as removing the unnecessary lines and spaces, and renaming the variables or properties. However, more targeted transformations also exist and can only be applied to certain inputs. For instance, transforming numbers to expressions can only be applied to numbers. If the code snippet does not contain any numbers, then it cannot be applied. The same is valid for splitting string literals into chunks of a certain length. If the original code to obfuscate does not include any string literals, then this obfuscation cannot be applied. A challenge that arises from this approach is building examples that are relevant to all code snippets; for this reason, we decided to combine at least two transformations in the examples used to conduct the study.

We attempted to add examples of high obfuscation as well, but this resulted in the prompt being too long, causing the model to throw an error. This is a significant limitation, as excessively long obfuscation examples may not be processable. They must be built in a way that includes sufficient obfuscation transformations, or the prompt could be split into several prompts, leading to a more systematic approach that is completed in several steps. A few of the settings we have enabled when building the examples include: compact, which removed all unnecessary spaces and new lines; hexadecimal or mangled name generators, which rename the variables and properties using hexadecimal values or English alphabetical lower case letters; the split strings setting splits string literals in smaller strings of a preset or custom length; numbers to expression setting rewrites numbers into equivalent expressions using arithmetical operators. A more advanced transformation is control flow flattening, which alters the code's structure.

5.3.2 External Validity

Due to the non-deterministic nature of LLMs, we expect slightly different results in a possible replication of our experiment. We aimed to mitigate this by introducing iterations. Each code snippet in the dataset was obfuscated using the same prompt ten times (iterations). However, results may differ significantly, even when using the same model with the prompts we crafted. We expect that further improvements in the prompt engineering technique of choice can have a positive effect on the LLM obfuscations.

The existing obfuscation solutions provide CI/CD pipeline integration throughout the project life cycle, which requires obfuscation across multiple files and modules, including those with and without dependencies, as well as internal and third-party modules. Additionally, for code that is not covered by any tests (such as unit tests or end-to-end tests), it is challenging to determine, without running any tests, whether the obfuscation will maintain the intended code behavior. When not entirely cor-

rect, the obfuscation can alter the code, causing significant problems in production environments. For this, code obfuscation using LLM needs to be studied further to make it suitable for real-world projects.

Code obfuscation can be used for different purposes. One would be hiding or camouflaging malicious code. If an LLM is capable of obfuscating malicious code using already existing approaches, but also discovering new patterns, this can be a cybersecurity threat [12].

6

Conclusion

In this study, we explored the capabilities of an LLM to obfuscate standalone JavaScript code, without pre-training, and without allowing the model to use any external resources or obfuscation tools. We ran a pre-study to find a suitable prompt engineering technique and to implement it optimally. During this step, we realized that the prompt can have a significant impact on the obfuscation results. To run the laboratory experiment, we developed an automated pipeline to establish a standardized approach for iterative processes and minimize errors.

Overall, the results show that an LLM can apply at least minimal obfuscation transformations on JavaScript code, but the prompt is the key element. Our customized Few-Shot prompts resulted in 72.8% correct iterations. However, many obfuscation results are quite similar to those obtained with a basic obfuscation setup provided by our baseline tool, JavaScript Obfuscator. The default setup implemented by this tool seems to perform a more advanced obfuscation.

Our evaluation methods proved to be suitable for our study. Code correctness is a priority in obfuscation. The obfuscated code must pass all tests to ensure that the intended behavior is maintained. Furthermore, a strong obfuscation changes the code entirely, while making it difficult to extract features or follow the structure. CodeBLEU is a code synthesis evaluation tool specifically designed to assess generated code. Although it does not explicitly measure readability, it can serve as a suitable indicator of how much the code has changed from its original form.

Our study is theoretical, but this area of study has great potential to become more practical as LLMs and our interaction with them evolve. One goal we mentioned in the abstract and introduction is that the LLMs could identify obfuscation patterns currently unknown or significantly difficult for existing deobfuscators to reverse-engineer. Our obfuscation focuses on relatively simple and standalone JavaScript code. For more web-oriented code, which includes code injected into HTML pages or JavaScript code used to build client-side applications using a framework such as React.js, the situation may be different, where deobfuscators may not be able to entirely reverse-engineer all parts of the code. As well as current obfuscators perform, no obfuscation is entirely irreversible. With further studies in this area, more advanced patterns may emerge, and state-of-the-art models should be employed in similar studies.

Although our results suggest that the LLM we selected can perform very basic obfuscation, an improved prompt engineering strategy could have a significant impact, which could be a topic for future work. LLMs show great potential in code-related tasks, and code obfuscation is a complex code synthesis task. The topic of code obfuscation using an LLM remains largely unexplored, and there are many possible future studies to consider. Another topic to explore further could be the obfuscation of Web page-specific JavaScript code, such as Document Object Model(DOM) manipulation. Our dataset consists of basic code that only implements different algorithms and data structure coding problems. We have no code specific to Web pages. In our laboratory experiment, we use only one LLM, and another future research work could be a comparison study of how different LLMs (including state-of-the-art) would obfuscate not only JavaScript code, but also other programming and scripting languages. Another potential direction would also be studying code deobfuscation using LLMs and their capabilities to reverse their obfuscations. Code obfuscation can be a challenging task, and a study on evaluation methods could also be suitable for future research.

Bibliography

[1] Fangzhou Wu, Qingzhao Zhang, Ati Priya Bajaj, Tiffany Bao, Ning Zhang, Ruoyu Wang, Chaowei Xiao, et al. Exploring the limits of chatgpt in software security applications. *arXiv preprint arXiv:2312.05275*, 2023.

[2] K. Martineau. " What is generative ai?". IBM, 2023. [Online]. Available: https://research.ibm.com/blog/what-is-generative-AI.

[3] Javascript obfuscator [online] available:. `https://www.npmjs.com/package/javascript-obfuscator`.

[4] Jsdefender [online] available:. `https://www.preemptive.com/products/jsdefender/`.

[5] Savio Antony Sebastian, Saurabh Malgaonkar, Paulami Shah, Mudit Kapoor, and Tanay Parekhji. A study & review on code obfuscation. In *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, pages 1–6. IEEE, 2016.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, 01 1997.

[7] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications, 2025.

[8] Constantinos Patsakis, Fran Casino, and Nikolaos Lykousas. Assessing llms in malicious code deobfuscation of real-world malware campaigns, 2024.

[9] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[10] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 1(2012):1–27, 2012.

[11] Zoltán Ságodi, István Siket, and Rudolf Ferenc. Methodology for code synthesis

evaluation of llms presented by a case study of chatgpt and copilot. *IEEE Access*, PP:1–1, 01 2024.

[12] Seyedreza Mohseni, Seyedali Mohammadi, Deepa Tilwani, Yash Saxena, Gerald Ketu Ndawula, Sriram Vema, Edward Raff, and Manas Gaur. Can llms obfuscate code? a systematic analysis of large language models into assembly code obfuscation, 2025.

[13] Sampsa Rauti and Ville Leppänen. A comparison of online javascript obfuscators. In *2018 International Conference on Software Security and Assurance (ICSSA)*, pages 7–12, 2018.

[14] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[15] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[16] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code, 2023.

[17] Klaas-Jan Stol and Brian Fitzgerald. The abc of software engineering research. *ACM Trans. Softw. Eng. Methodol.*, 27(3), September 2018.

[18] Alvis cluster, the naiss resource dedicated for artificial intelligence and machine learning research [online]. `https://www.c3se.chalmers.se/about/Alvis/`.

[19] Leetcode [online] available:. `https://leetcode.com/`.

[20] Huggingface llama-3.1-8b-instruct [online] available:. `https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct`.

[21] Huggingface [online] available:. `https://huggingface.co/`.

[22] Obfuscator.io [online] available:. `https://obfuscator.io/#preset-options`.

[23] Npm [online] available:. `https://www.npmjs.com/`.

[24] Codebleu python package implementation [online] available:. `https://pypi.org/project/codebleu/`.

[25] Leetcode coding problem: Search insert position [online] available:. `https://leetcode.com/problems/search-insert-position/description/`.

[26] Leetcode coding problem: Combination sum [online] available:. `https://leetcode.com/problems/combination-sum/description/`.

A

Appendix - Dataset code snippets by difficulty level

A.1 Difficulty level - Easy

Problem statement: Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You must write an algorithm with O(log n) runtime complexity [25].

Input 1	Input 2	Expected output
[1, 3, 5, 6]	5	2
[1, 3, 5, 6]	2	1
[1, 3, 5, 6]	7	4

Table A.1:Test suite for id_7

```
var id_7 = function (nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left <= right) {
        let mid = Math.floor((left + right) / 2);

        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return left;
};

module.exports = { id_7 };
```

A.2 Difficulty level - Medium

Problem statement: You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise). You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation [26].

Input 1	Input 2	Expected output
[2, 3, 6, 7]	7	[[2, 2, 3] , [7]]
[2, 3, 5]	8	[[2, 2, 2, 2] , [2, 3, 3] , [3, 5]]
[2]	1	[]

Table A.2:Test suite for id_15

```
var id_15 = function (candidates, target) {
    const res = [];

    function makeCombination(idx, comb, total) {
        if (total === target) {
            res.push([...comb]);
            return;
        }

        if (total > target || idx >= candidates.length) {
            return;
        }

        comb.push(candidates[idx]);
        makeCombination(idx, comb, total + candidates[idx]);
        comb.pop();
        makeCombination(idx + 1, comb, total);
    }

    makeCombination(0, [], 0);
    return res;
};

module.exports = { id_15 };
```

A.3 Difficulty level - High

Problem statement: Given a string s, return whether s is a valid number.

For example, all the following are valid numbers: "2", "0089", "-0.1", "+3.14", "4.", "-.9", "2e10", "-90E3", "3e+7", "+6e-1", "53.5e93", "-123.456e789", while the following are not valid numbers: "abc", "1a", "1e", "e3", "99e2.5", "–6", "-+3", "95a54e53".

Formally, a valid number is defined using one of the following definitions:

1. An integer number followed by an optional exponent.

2. A decimal number followed by an optional exponent.

An integer number is defined with an optional sign '-' or '+' followed by digits. A decimal number is defined with an optional sign '-' or '+' followed by one of the following definitions:

1. Digits followed by a dot '.'.

2. Digits followed by a dot '.' followed by digits.

3. A dot '.' followed by digits.

An exponent is defined with an exponent notation 'e' or 'E' followed by an integer number. The digits are defined as one or more digits.

Input 1	Expected output
"0"	true
"e"	false
"."	false

Table A.3:Test suite for id_23

```
var id_23 = function (S) {
    let exp = false, sign = false, num = false, dec = false
    for (let c of S)
        if (c >= '0' && c <= '9') num = true
        else if (c === 'e' || c === 'E')
            if (exp || !num) return false
            else exp = true, sign = false, num = false, dec = false
        else if (c === '+' || c === '-')
            if (sign || num || dec) return false
            else sign = true
        else if (c === '.')
            if (dec || exp) return false
            else dec = true
        else return false
    return num
};

module.exports = { id_23 };
```

B

Appendix - Few-Shot Prompt

B.0.1 Prompt for a code snippet corresponding to the easy difficulty level coding problem

```
var twoSum = function(nums, target) {
    const pairIdx = {};

    for (let i = 0; i < nums.length; i++) {
        const num = nums[i];
        if (target - num in pairIdx) {
            return [i, pairIdx[target - num]];
        }
        pairIdx[num] = i;
    }
    return pairIdx;
};
module.exports = {twoSum};
The code above can be obfuscated in several ways, such as the
   following examples:
Example 1:
    var twoSum=function(_0x2d1fc5,_0x227c05){const _0x2f1f1e={};for
       (let _0x2d9100=0x0;_0x2d9100<_0x2d1fc5['length'];_0x2d9100
       ++){const _0x5dc20f=_0x2d1fc5[_0x2d9100];if(_0x227c05-
       _0x5dc20f in _0x2f1f1e){return[_0x2d9100,_0x2f1f1e[_0x227c05
       -_0x5dc20f]];}_0x2f1f1e[_0x5dc20f]=_0x2d9100;}return
       _0x2f1f1e;};module['exports']={'twoSum':twoSum};
Example 2:
    var twoSum=function(a,b){const c={};for(let d=0x0;d<a['length'
       ];d++){const e=a[d];if(b-e in c){return[d,c[b-e]];}c[e]=d;}
       return c;};module['exports']={'twoSum':twoSum};
Example 3:
    const i=b;var twoSum=function(c,d){const h=b;const e={};for(let
        f=0x0;f<c[h(0x0)];f++){const g=c[f];if(d-g in e){return[f,e
       [d-g]];}e[g]=f;}return e;};function a(){const j=['length','
       exports'];a=function(){return j;};return a();}function b(c,d
       ){const e=a();b=function(f,g){f=f-0x0;let h=e[f];return h;};
       return b(c,d);}module[i(0x1)]={'twoSum':twoSum};

    Considering the examples above and without using any external
       tools or APIs, obfuscate the following JavaScript code var
       id_7 = function (nums, target) {
    let left = 0;
```

```
    let right = nums.length - 1;

    while (left <= right) {
        let mid = Math.floor((left + right) / 2);

        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return left;
};

module.exports = { id_7 }; and return only one obfuscated code
   snippet. Your output should start with ```javascript and end
   with```
```

C

Appendix - CodeBLEU Scores for LLm-obfuscated code for each configuration

Id	Min	Max	Mean	SD	Median
1	0.308	0.308	0.308	NA	0.308
2	0.169	0.227	0.196	0.029	0.191
3	0.166	0.377	0.275	0.065	0.264
4	0.287	0.473	0.342	0.055	0.325
5	0.236	0.352	0.288	0.044	0.276
6	0.223	0.357	0.271	0.037	0.259
7	0.246	0.399	0.29	0.055	0.258
8	0.298	0.367	0.342	0.028	0.36
9	0.263	0.354	0.307	0.033	0.308
10	0.28	0.324	0.302	0.031	0.302

Table C.1: CodeBLEU - LLM Obfuscation for Configuration 1 by ID iterations

Id	Min	Max	Mean	SD	Median
1	0.312	0.475	0.411	0.079	0.429
3	0.204	0.586	0.388	0.119	0.385
4	0.339	0.505	0.418	0.063	0.414
5	0.266	0.788	0.445	0.148	0.438
6	0.332	0.523	0.418	0.068	0.391
7	0.324	0.449	0.379	0.042	0.387
8	0.302	0.485	0.405	0.067	0.39
9	0.157	0.395	0.302	0.084	0.321
10	0.283	0.54	0.388	0.101	0.371

Table C.2: CodeBLEU - LLM Obfuscation for Configuration 2 by ID iterations

Id	Min	Max	Mean	SD	Median
11	0.294	0.417	0.356	0.057	0.377
12	0.277	0.396	0.325	0.045	0.316
13	0.541	0.597	0.564	0.017	0.561
14	0.354	0.524	0.436	0.066	0.437
15	0.327	0.543	0.45	0.092	0.469
16	0.313	0.357	0.336	0.013	0.337
17	0.297	0.597	0.478	0.116	0.492
18	0.276	0.538	0.399	0.091	0.391
19	0.298	0.406	0.343	0.036	0.342
20	0.282	0.704	0.402	0.157	0.327

Table C.3: CodeBLEU - LLM Obfuscation for Configuration 3 by ID iterations

Id	Min	Max	Mean	SD	Median
11	0.287	0.439	0.366	0.05	0.379
12	0.257	0.638	0.468	0.168	0.462
13	0.516	0.635	0.565	0.044	0.549
14	0.411	0.563	0.477	0.056	0.487
15	0.364	0.79	0.466	0.13	0.42
16	0.244	0.406	0.339	0.054	0.339
17	0.444	0.817	0.56	0.132	0.52
18	0.299	0.603	0.393	0.1	0.382
19	0.335	0.578	0.448	0.075	0.444
20	0.284	0.654	0.372	0.118	0.334

Table C.4: CodeBLEU - LLM Obfuscation for Configuration 4 by ID iterations

Id	Min	Max	Mean	SD	Median
21	0.377	0.909	0.733	0.171	0.742
22	0.345	0.652	0.441	0.108	0.412
23	0.273	0.574	0.477	0.114	0.508
24	0.2	0.462	0.343	0.116	0.372
25	0.354	0.536	0.432	0.078	0.413
26	0.329	0.622	0.443	0.129	0.393
28	0.467	0.822	0.627	0.116	0.621
29	0.461	0.597	0.524	0.049	0.525
30	0.29	0.38	0.33	0.032	0.32

Table C.5: CodeBLEU - LLM Obfuscation for Configuration 5 by ID iterations

D

Appendix - CodeBLEU Scores for JS Obfuscator code for each obfuscation level

Code difficulty	Min	Max	Mean	SD	Median
Easy	0.252	0.439	0.362	0.058	0.357
Medium	0.237	0.563	0.326	0.09	0.32
High	0.139	0.324	0.256	0.06	0.279

Table D.1: CodeBLEU - JavaScript Obfuscator per difficulty - BASIC 1

Code difficulty	Min	Max	Mean	SD	Median
Easy	0.205	0.404	0.293	0.059	0.296
Medium	0.221	0.338	0.259	0.036	0.257
High	0.109	0.274	0.214	0.06	0.224

Table D.2: CodeBLEU - JavaScript Obfuscator per difficulty - BASIC 2

Code difficulty	Min	Max	Mean	SD	Median
Easy	0.185	0.299	0.245	0.032	0.245
Medium	0.132	0.301	0.224	0.049	0.225
High	0.106	0.243	0.183	0.052	0.205

Table D.3: CodeBLEU - JavaScript Obfuscator per difficulty - DEFAULT

Code difficulty	Min	Max	Mean	SD	Median
Easy	0.179	0.313	0.242	0.041	0.236
Medium	0.181	0.295	0.213	0.033	0.208
High	0.092	0.22	0.172	0.046	0.19

Table D.4: CodeBLEU - JavaScript Obfuscator per difficulty - MEDIUM

Code difficulty	Min	Max	Mean	SD	Median
Easy	0.183	0.305	0.225	0.038	0.222
Medium	0.185	0.318	0.22	0.043	0.196
High	0.081	0.261	0.184	0.06	0.196

Table D.5: CodeBLEU - JavaScript Obfuscator per difficulty - HIGH

E

Appendix - JavaScript Obfuscations for ID 9 and ID 21

E.1 ID 9

E.1.1 JavaScript Obfuscator - Medium, CodeBLEU = 0.179

```
function _0x29ff(_0x7a4687,_0x282f27){const _0x432ee1=_0x432e();
    return _0x29ff=function(_0x29ff58,_0x3b7f9f){_0x29ff58=_0x29ff58
    -0x111;let _0x43dc0c=_0x432ee1[_0x29ff58];return _0x43dc0c;},
    _0x29ff(_0x7a4687,_0x282f27);}const _0x4f7b6b=_0x29ff;function
    _0x432e(){const _0x33da32=['5310253GfedUl','5871036mhxERD','7096
    ieGaeo','36FwvFYp','94760OnlkrQ','substring','sort','11rznbpH','
    1sKKijc','426266cvZVAc','31420GMccNW','438RBWWxI','245cPekNw','
    exports','1386590SAUmqg','length','543YGIyqL'];_0x432e=function
    (){return _0x33da32;};return _0x432e();}(function(_0x384892,
    _0x1fbc07){const _0x5715b3=_0x29ff,_0x351598=_0x384892();while
    (!![]){try{const _0x18af11=parseInt(_0x5715b3(0x111))/0x1*(
    parseInt(_0x5715b3(0x112))/0x2)+-parseInt(_0x5715b3(0x119))/0x3
    *(-parseInt(_0x5715b3(0x11c))/0x4)+-parseInt(_0x5715b3(0x113))/0
    x5*(parseInt(_0x5715b3(0x114))/0x6)+-parseInt(_0x5715b3(0x115))
    /0x7*(parseInt(_0x5715b3(0x11e))/0x8)+-parseInt(_0x5715b3(0x11d)
    )/0x9*(-parseInt(_0x5715b3(0x117))/0xa)+-parseInt(_0x5715b3(0
    x121))/0xb*(-parseInt(_0x5715b3(0x11b))/0xc)+-parseInt(_0x5715b3
    (0x11a))/0xd;if(_0x18af11===_0x1fbc07)break;else _0x351598['push
    '](_0x351598['shift']());}catch(_0x1e330a){_0x351598['push'](
    _0x351598['shift']());}}}(_0x432e,0x48588));var id_9=function(
    _0x48a2da){const _0x404d31=_0x29ff;let _0x42c2d9='';_0x48a2da[
    _0x404d31(0x120)]();const _0x58b4db=_0x48a2da[0x0],_0x1d16a1=
    _0x48a2da[_0x48a2da['length']-0x1],_0x4edb01=_0x58b4db[_0x404d31
    (0x118)];for(let _0x8045bc=0x0;_0x8045bc<_0x4edb01;_0x8045bc++){
    if(_0x58b4db['substring'](_0x8045bc,_0x8045bc+0x1)===_0x1d16a1[
    _0x404d31(0x11f)](_0x8045bc,_0x8045bc+0x1))_0x42c2d9+=_0x58b4db[
    _0x404d31(0x11f)](_0x8045bc,_0x8045bc+0x1);else return _0x42c2d9
    ;}return _0x42c2d9;};module[_0x4f7b6b(0x116)]={'id_9':id_9};
```

E.1.2 JavaScript Obfuscator - High, CodeBLEU = 0.222

```
const _0x413fd2=_0x43ba;(function(_0x6c99a9,_0x16bd01){const
    _0x2fb135=_0x43ba,_0x19b95d=_0x6c99a9();while(!![]){try{const
    _0x1a3182=parseInt(_0x2fb135(0xc5))/0x1*(parseInt(_0x2fb135(0xc8
```

```
))/0x2)+parseInt(_0x2fb135(0xcf))/0x3*(parseInt(_0x2fb135(0xc7))
/0x4)+parseInt(_0x2fb135(0xcc))/0x5*(-parseInt(_0x2fb135(0xc6))
/0x6)+parseInt(_0x2fb135(0xc4))/0x7+-parseInt(_0x2fb135(0xc9))/0
x8+-parseInt(_0x2fb135(0xce))/0x9+parseInt(_0x2fb135(0xd0))/0xa
*(parseInt(_0x2fb135(0xca))/0xb);if(_0x1a3182===_0x16bd01)break;
else _0x19b95d['push'](_0x19b95d['shift']());}catch(_0x5352ea){
_0x19b95d['push'](_0x19b95d['shift']());}}}(_0x22dc,0xe4bca));
function _0x43ba(_0x29ece5,_0x494fa8){const _0x22dc0b=_0x22dc();
return _0x43ba=function(_0x43ba6a,_0x2e13a9){_0x43ba6a=_0x43ba6a
-0xc3;let _0x40d5ff=_0x22dc0b[_0x43ba6a];return _0x40d5ff;},
_0x43ba(_0x29ece5,_0x494fa8);}var id_9=function(_0x320678){const
 _0x44fa5b=_0x43ba;let _0x3f84e0='';_0x320678[_0x44fa5b(0xcb)]()
;const _0x1d32c9=_0x320678[0x0],_0x31573c=_0x320678[_0x320678['
length']-0x1],_0x1d757e=_0x1d32c9[_0x44fa5b(0xc3)];for(let
_0x3a812c=0x0;_0x3a812c<_0x1d757e;_0x3a812c++){if(_0x1d32c9[
_0x44fa5b(0xcd)](_0x3a812c,_0x3a812c+0x1)===_0x31573c['substring
'](_0x3a812c,_0x3a812c+0x1))_0x3f84e0+=_0x1d32c9[_0x44fa5b(0xcd)
](_0x3a812c,_0x3a812c+0x1);else return _0x3f84e0;}return
_0x3f84e0;};module[_0x413fd2(0xd1)]={'id_9':id_9};function
_0x22dc(){const _0xf70dd3=['length','10530695ZIQmxC','288217
FRrnhn','35556jAOlZC','12znwKOc','4sikVvQ','13382288vsDSAC','
9980454XlbpgP','sort','65THOZst','substring','15669684GXbbIV','
1439673DhMHMr','10suTWAa','exports'];_0x22dc=function(){return
_0xf70dd3;};return _0x22dc();}
```

E.2 ID 17

E.2.1 JavaScript Obfuscator - Basic 1, CodeBLEU = 0.237

```
1  var id_17 = function (_0x1ff9ad, _0x43d822) {
2      function _0x20053d(_0x2e041d, _0x42c472) {
3          if (_0x2e041d === 0x0) {
4              return 0x0;
5          }
6          if (_0x42c472 === 0x0) {
7              return 0x1;
8          }
9          let _0x577341 = _0x20053d(_0x2e041d, Math['floor'](
               _0x42c472 / 0x2));
10         _0x577341 = _0x577341 * _0x577341;
11         if (_0x42c472 % 0x2 === 0x1) {
12             return parseFloat(_0x577341 * _0x2e041d)['toFixed'](0x5
                   );
13         }
14         return parseFloat(_0x577341)['toFixed'](0x5);
15     }
16     let _0x29bb0e = _0x20053d(_0x1ff9ad, Math['abs'](_0x43d822));
17     if (_0x43d822 >= 0x0) {
18         return parseFloat(_0x29bb0e)['toFixed'](0x5);
19     }
20     return parseFloat(0x1 / _0x29bb0e)['toFixed'](0x5);
21 };
22 module['exports'] = { 'id_17': id_17 };
```

E.2.2 JavaScript Obfuscator - Default, CodeBLEU = 0.181

```
const _0x414b33=_0x48e8;(function(_0x4c1228,_0x213da9){const
    _0x39de15=_0x48e8,_0x56adf5=_0x4c1228();while(!![]){try{const
    _0x17f349=-parseInt(_0x39de15(0xa7))/0x1*(-parseInt(_0x39de15(0
    xa5))/0x2)+parseInt(_0x39de15(0xaf))/0x3*(parseInt(_0x39de15(0
    xad))/0x4)+parseInt(_0x39de15(0xa9))/0x5+parseInt(_0x39de15(0xa8
    ))/0x6*(-parseInt(_0x39de15(0xa6))/0x7)+parseInt(_0x39de15(0xb2)
    )/0x8*(-parseInt(_0x39de15(0xab))/0x9)+parseInt(_0x39de15(0xb1))
    /0xa*(parseInt(_0x39de15(0xb3))/0xb)+-parseInt(_0x39de15(0xae))
    /0xc*(parseInt(_0x39de15(0xac))/0xd);if(_0x17f349===_0x213da9)
    break;else _0x56adf5['push'](_0x56adf5['shift']());}catch(
    _0x373119){_0x56adf5['push'](_0x56adf5['shift']());}}}(_0x3715,0
    x881fe));function _0x3715(){const _0x42b70d=['3189735OfyfUi','
    abs','9622287yNkPBV','39871saSEKD','556940QOaYcP','948UJascS','3
    QVaLAO','toFixed','2670Teirvp','8QgILys','33132qTkOvG','exports'
    ,'263918qKvIca','2605253MmmuuR','5lRqrTF','6AEqWmN'];_0x3715=
    function(){return _0x42b70d;};return _0x3715();}function _0x48e8
    (_0x5770b6,_0x5c979e){const _0x371514=_0x3715();return _0x48e8=
    function(_0x48e84b,_0x25deff){_0x48e84b=_0x48e84b-0xa4;let
    _0x4310b9=_0x371514[_0x48e84b];return _0x4310b9;},_0x48e8(
    _0x5770b6,_0x5c979e);}var id_17=function(_0x1afa18,_0x217e8a){
    const _0x561ab8=_0x48e8;function _0x40d052(_0xa62f0,_0x3d1d3e){
    const _0x3e391f=_0x48e8;if(_0xa62f0===0x0)return 0x0;if(
    _0x3d1d3e===0x0)return 0x1;let _0x119630=_0x40d052(_0xa62f0,Math
    ['floor'](_0x3d1d3e/0x2));_0x119630=_0x119630*_0x119630;if(
    _0x3d1d3e%0x2===0x1)return parseFloat(_0x119630*_0xa62f0)[
    _0x3e391f(0xb0)](0x5);return parseFloat(_0x119630)[_0x3e391f(0
    xb0)](0x5);}let _0x5b3174=_0x40d052(_0x1afa18,Math[_0x561ab8(0
    xaa)](_0x217e8a));if(_0x217e8a>=0x0)return parseFloat(_0x5b3174)
    [_0x561ab8(0xb0)](0x5);return parseFloat(0x1/_0x5b3174)['toFixed
    '](0x5);};module[_0x414b33(0xa4)]={'id_17':id_17};
```

E.2.3 JavaScript Obfuscator - Medium, CodeBLEU = 0.181

```
const _0x403857=_0x59b5;(function(_0x563740,_0xcbe392){const
    _0xf2e78a=_0x59b5,_0x92e762=_0x563740();while(!![]){try{const
    _0x450955=-parseInt(_0xf2e78a(0xe0))/0x1+parseInt(_0xf2e78a(0xe5
    ))/0x2+-parseInt(_0xf2e78a(0xe6))/0x3*(parseInt(_0xf2e78a(0xe1))
    /0x4)+-parseInt(_0xf2e78a(0xe8))/0x5*(parseInt(_0xf2e78a(0xeb))
    /0x6)+-parseInt(_0xf2e78a(0xe9))/0x7*(-parseInt(_0xf2e78a(0xee))
    /0x8)+-parseInt(_0xf2e78a(0xdf))/0x9*(parseInt(_0xf2e78a(0xed))
    /0xa)+parseInt(_0xf2e78a(0xec))/0xb*(parseInt(_0xf2e78a(0xe7))/0
    xc);if(_0x450955===_0xcbe392)break;else _0x92e762['push'](
    _0x92e762['shift']());}catch(_0x2ed86c){_0x92e762['push'](
    _0x92e762['shift']());}}}(_0x13e2,0xd21f1));var id_17=function(
    _0x4d4c59,_0x4e530f){const _0x4ae5fb=_0x59b5;function _0x2f8a57(
    _0x4c361b,_0x2965c0){const _0x42b9e7=_0x59b5;if(_0x4c361b===0x0)
    return 0x0;if(_0x2965c0===0x0)return 0x1;let _0x4aaf56=_0x2f8a57
    (_0x4c361b,Math[_0x42b9e7(0xe2)](_0x2965c0/0x2));_0x4aaf56=
    _0x4aaf56*_0x4aaf56;if(_0x2965c0%0x2===0x1)return parseFloat(
    _0x4aaf56*_0x4c361b)['toFixed'](0x5);return parseFloat(_0x4aaf56
    )['toFixed'](0x5);}let _0x12f121=_0x2f8a57(_0x4d4c59,Math[
    _0x4ae5fb(0xe3)](_0x4e530f));if(_0x4e530f>=0x0)return parseFloat
    (_0x12f121)[_0x4ae5fb(0xea)](0x5);return parseFloat(0x1/
    _0x12f121)[_0x4ae5fb(0xea)](0x5);};function _0x59b5(_0x2c71df,
```

```
    _0xeaa7b6){const _0x13e21d=_0x13e2();return _0x59b5=function(
    _0x59b593,_0x35a1f9){_0x59b593=_0x59b593-0xdf;let _0x265b09=
    _0x13e21d[_0x59b593];return _0x265b09;},_0x59b5(_0x2c71df,
    _0xeaa7b6);}function _0x13e2(){const _0x142f04=['55uivNgN','7670
    OndDIv','547680opCSaX','12690IUDFYc','886392IwcyKI','6697892
    XDqbnV','floor','abs','exports','2614318DFCbbF','3xMpOxD','
    9938076TRYHqs','1523590FqvWzW','28klzhTE','toFixed','24kvIoYy'];
    _0x13e2=function(){return _0x142f04;};return _0x13e2();}module[
    _0x403857(0xe4)]={'id_17':id_17};
```

E.2.4 JavaScript Obfuscator - High, CodeBLEU - 0.190

```
1 (function(_0x4966bc,_0x5cd496){const _0x3302a9=_0x1b47,_0x560bad=
    _0x4966bc();while(!![]){try{const _0x1ce061=parseInt(_0x3302a9(0
    x17f))/0x1*(-parseInt(_0x3302a9(0x179))/0x2)+parseInt(_0x3302a9
    (0x180))/0x3+-parseInt(_0x3302a9(0x17d))/0x4*(parseInt(_0x3302a9
    (0x178))/0x5)+parseInt(_0x3302a9(0x177))/0x6+-parseInt(_0x3302a9
    (0x17b))/0x7+parseInt(_0x3302a9(0x176))/0x8*(-parseInt(_0x3302a9
    (0x17c))/0x9)+parseInt(_0x3302a9(0x17a))/0xa;if(_0x1ce061===
    _0x5cd496)break;else _0x560bad['push'](_0x560bad['shift']());}
    catch(_0x4e8eb9){_0x560bad['push'](_0x560bad['shift']());}}}(
    _0x52b8,0xa37dc));function _0x1b47(_0x50af57,_0x3c3ff1){const
    _0x52b8ed=_0x52b8();return _0x1b47=function(_0x1b472f,_0x5b1da4)
    {_0x1b472f=_0x1b472f-0x174;let _0x26259b=_0x52b8ed[_0x1b472f];
    return _0x26259b;},_0x1b47(_0x50af57,_0x3c3ff1);}function
    _0x52b8(){const _0x4c5dd0=['30365wimisn','3210852wkdSFH','
    toFixed','abs','112vUtzzG','2597676tzcAZG','25suzpVX','4vLpSsO',
    '19112760WvOVGP','7099925YaNaSc','318726UnjycO','939236TyEDBo','
    floor'];_0x52b8=function(){return _0x4c5dd0;};return _0x52b8();}
    var id_17=function(_0x49f99d,_0x25d3d2){const _0x5e805a=_0x1b47;
    function _0x54c8f7(_0x5d64f7,_0x1743aa){const _0x3eb452=_0x1b47;
    if(_0x5d64f7===0x0)return 0x0;if(_0x1743aa===0x0)return 0x1;let
    _0x2097e7=_0x54c8f7(_0x5d64f7,Math[_0x3eb452(0x17e)](_0x1743aa/0
    x2));_0x2097e7=_0x2097e7*_0x2097e7;if(_0x1743aa%0x2===0x1)return
     parseFloat(_0x2097e7*_0x5d64f7)[_0x3eb452(0x174)](0x5);return
    parseFloat(_0x2097e7)['toFixed'](0x5);}let _0x1f3bc6=_0x54c8f7(
    _0x49f99d,Math[_0x5e805a(0x175)](_0x25d3d2));if(_0x25d3d2>=0x0)
    return parseFloat(_0x1f3bc6)['toFixed'](0x5);return parseFloat(0
    x1/_0x1f3bc6)['toFixed'](0x5);};module['exports']={'id_17':id_17
    };
```

E.3 ID 21

E.3.1 JavaScript Obfuscator - Basic 2, CodeBLEU = 0.252

```
1 var id_21=function(_0x2e89af,_0x2073d5){const _0x19e67b=Array['from
    ']({'length':_0x2e89af['length']+0x1},()=>Array['from']({'length
    ':_0x2073d5['length']+0x1}));_0x19e67b[_0x2e89af['length']][
    _0x2073d5['length']]=!![];const _0x361aaf=(_0x2fb62f,_0x13a242)
    =>{if(_0x19e67b[_0x2fb62f][_0x13a242]!==undefined)return
    _0x19e67b[_0x2fb62f][_0x13a242];const _0x48ef0d=_0x2fb62f<
    _0x2e89af['length']&&(_0x2073d5[_0x13a242]==='.'||_0x2e89af[
    _0x2fb62f]===_0x2073d5[_0x13a242]);if(_0x13a242+0x1<_0x2073d5['
    length']&&_0x2073d5[_0x13a242+0x1]==='*'){_0x19e67b[_0x2fb62f][
```

```
    _0x13a242]=_0x361aaf(_0x2fb62f,_0x13a242+0x2)||_0x48ef0d&&
    _0x361aaf(_0x2fb62f+0x1,_0x13a242);}else{_0x19e67b[_0x2fb62f][
    _0x13a242]=_0x48ef0d&&_0x361aaf(_0x2fb62f+0x1,_0x13a242+0x1);}
    return _0x19e67b[_0x2fb62f][_0x13a242];};return _0x361aaf(0x0,0
    x0);};module['exports']={'id_21':id_21};
```

E.3.2 JavaScript Obfuscator - Default, CodeBLEU = 0.198

```
1 function _0x3c68(_0x7d8848,_0x3edfc3){const _0x46fc9b=_0x46fc();
    return _0x3c68=function(_0x3c689d,_0xb1f300){_0x3c689d=_0x3c689d
    -0x1d5;let _0x3ca1ad=_0x46fc9b[_0x3c689d];return _0x3ca1ad;},
    _0x3c68(_0x7d8848,_0x3edfc3);}const _0xd21947=_0x3c68;(function(
    _0x33a26e,_0x5c04b2){const _0x181012=_0x3c68,_0x293ce0=_0x33a26e
    ();while(!![]){try{const _0x586ed8=-parseInt(_0x181012(0x1e0))/0
    x1+-parseInt(_0x181012(0x1dc))/0x2+-parseInt(_0x181012(0x1db))/0
    x3*(-parseInt(_0x181012(0x1d7))/0x4)+parseInt(_0x181012(0x1da))
    /0x5+parseInt(_0x181012(0x1d9))/0x6+parseInt(_0x181012(0x1dd))/0
    x7*(parseInt(_0x181012(0x1d5))/0x8)+parseInt(_0x181012(0x1de))/0
    x9;if(_0x586ed8===_0x5c04b2)break;else _0x293ce0['push'](
    _0x293ce0['shift']());}catch(_0x372f35){_0x293ce0['push'](
    _0x293ce0['shift']());}}}(_0x46fc,0xd3998));var id_21=function(
    _0x4badf2,_0x17b75c){const _0x44622d=_0x3c68,_0x231232=Array[
    _0x44622d(0x1d8)]({'length':_0x4badf2['length']+0x1},()=>Array['
    from']({'length':_0x17b75c[_0x44622d(0x1d6)]+0x1}));_0x231232[
    _0x4badf2['length']][_0x17b75c['length']]=!![];const _0x2fe3f9=(
    _0x2cd6f6,_0x22acd7)=>{const _0x213a5c=_0x44622d;if(_0x231232[
    _0x2cd6f6][_0x22acd7]!==undefined)return _0x231232[_0x2cd6f6][
    _0x22acd7];const _0x3cafe2=_0x2cd6f6<_0x4badf2[_0x213a5c(0x1d6)
    ]&&(_0x17b75c[_0x22acd7]==='.'||_0x4badf2[_0x2cd6f6]===_0x17b75c
    [_0x22acd7]);return _0x22acd7+0x1<_0x17b75c[_0x213a5c(0x1d6)]&&
    _0x17b75c[_0x22acd7+0x1]==='*'?_0x231232[_0x2cd6f6][_0x22acd7]=
    _0x2fe3f9(_0x2cd6f6,_0x22acd7+0x2)||_0x3cafe2&&_0x2fe3f9(
    _0x2cd6f6+0x1,_0x22acd7):_0x231232[_0x2cd6f6][_0x22acd7]=
    _0x3cafe2&&_0x2fe3f9(_0x2cd6f6+0x1,_0x22acd7+0x1),_0x231232[
    _0x2cd6f6][_0x22acd7];};return _0x2fe3f9(0x0,0x0);};module[
    _0xd21947(0x1df)]={'id_21':id_21};function _0x46fc(){const
    _0x51bf02=['8498124GGsAZA','exports','61798MScrgV','8zOckly','
    length','8yyvxuJ','from','2194770oxXFqD','138095cPXyxQ','1898418
    chpeeT','3401154CZmvpQ','180775TXfPUP'];_0x46fc=function(){
    return _0x51bf02;};return _0x46fc();}
```

E.3.3 JavaScript Obfuscator - Medium, CodeBLEU = 0.182

```
1 function _0x4a15(){const _0x524056=['144218DoPIUq','3272812dtUWJR',
    'exports','126ZJPMkf','1548276bvDJhJ','146650PnwCyY','20223310
    LvGQmi','10BePEGV','7EQAAuq','473464zhWXZN','length','1951314
    rjhhEo','from'];_0x4a15=function(){return _0x524056;};return
    _0x4a15();}const _0x33ae2b=_0x5b6a;function _0x5b6a(_0x448942,
    _0x51311b){const _0x4a1544=_0x4a15();return _0x5b6a=function(
    _0x5b6a31,_0x9d4a7b){_0x5b6a31=_0x5b6a31-0x137;let _0x2e3e63=
    _0x4a1544[_0x5b6a31];return _0x2e3e63;},_0x5b6a(_0x448942,
    _0x51311b);}(function(_0x4ba339,_0x3b0e2b){const _0x3a5c8a=
    _0x5b6a,_0x5b146b=_0x4ba339();while(!![]){try{const _0x662a1a=
    parseInt(_0x3a5c8a(0x13d))/0x1*(-parseInt(_0x3a5c8a(0x137))/0x2)
    +-parseInt(_0x3a5c8a(0x13b))/0x3+-parseInt(_0x3a5c8a(0x13e))/0x4
```

```
    +parseInt(_0x3a5c8a(0x142))/0x5+parseInt(_0x3a5c8a(0x141))/0x6
    *(-parseInt(_0x3a5c8a(0x138))/0x7)+-parseInt(_0x3a5c8a(0x139))/0
    x8*(-parseInt(_0x3a5c8a(0x140))/0x9)+parseInt(_0x3a5c8a(0x143))
    /0xa;if(_0x662a1a===_0x3b0e2b)break;else _0x5b146b['push'](
    _0x5b146b['shift']());}catch(_0x42c95e){_0x5b146b['push'](
    _0x5b146b['shift']());}}}(_0x4a15,0x6993e));var id_21=function(
    _0x1c619d,_0x3be89d){const _0x5ab267=_0x5b6a,_0x13d128=Array[
    _0x5ab267(0x13c)]({'length':_0x1c619d[_0x5ab267(0x13a)]+0x1},()
    =>Array['from']({'length':_0x3be89d['length']+0x1}));_0x13d128[
    _0x1c619d[_0x5ab267(0x13a)]][_0x3be89d['length']]=!![];const
    _0x1f4d58=(_0x1a1521,_0x1de0ab)=>{if(_0x13d128[_0x1a1521][
    _0x1de0ab]!==undefined)return _0x13d128[_0x1a1521][_0x1de0ab];
    const _0x1a28a2=_0x1a1521<_0x1c619d['length']&&(_0x3be89d[
    _0x1de0ab]==='.'||_0x1c619d[_0x1a1521]===_0x3be89d[_0x1de0ab]);
    return _0x1de0ab+0x1<_0x3be89d['length']&&_0x3be89d[_0x1de0ab+0
    x1]==='*'?_0x13d128[_0x1a1521][_0x1de0ab]=_0x1f4d58(_0x1a1521,
    _0x1de0ab+0x2)||_0x1a28a2&&_0x1f4d58(_0x1a1521+0x1,_0x1de0ab):
    _0x13d128[_0x1a1521][_0x1de0ab]=_0x1a28a2&&_0x1f4d58(_0x1a1521+0
    x1,_0x1de0ab+0x1),_0x13d128[_0x1a1521][_0x1de0ab];};return
    _0x1f4d58(0x0,0x0);};module[_0x33ae2b(0x13f)]={'id_21':id_21};
```

E.3.4 JavaScript Obfuscator - High, CodeBLEU = 0.185

```
1 function _0x5d9e(_0x321706,_0x30cd1f){const _0x3ddd7d=_0x3ddd();
    return _0x5d9e=function(_0x5d9e9f,_0x440baa){_0x5d9e9f=_0x5d9e9f
    -0x19d;let _0x2bfe9b=_0x3ddd7d[_0x5d9e9f];return _0x2bfe9b;},
    _0x5d9e(_0x321706,_0x30cd1f);}(function(_0x3553a9,_0x3ede52){
    const _0x39f78f=_0x5d9e,_0x38dbac=_0x3553a9();while(!![]){try{
    const _0x327ff3=parseInt(_0x39f78f(0x1a6))/0x1*(parseInt(
    _0x39f78f(0x1a9))/0x2)+parseInt(_0x39f78f(0x1a3))/0x3*(parseInt(
    _0x39f78f(0x19f))/0x4)+-parseInt(_0x39f78f(0x1a5))/0x5*(-
    parseInt(_0x39f78f(0x1a1))/0x6)+-parseInt(_0x39f78f(0x19d))/0x7
    +-parseInt(_0x39f78f(0x19e))/0x8+parseInt(_0x39f78f(0x1a2))/0x9
    *(parseInt(_0x39f78f(0x1a7))/0xa)+parseInt(_0x39f78f(0x1a0))/0xb
    ;if(_0x327ff3===_0x3ede52)break;else _0x38dbac['push'](_0x38dbac
    ['shift']());}catch(_0x4727ce){_0x38dbac['push'](_0x38dbac['
    shift']());}}}(_0x3ddd,0x7dc26));var id_21=function(_0x22613b,
    _0x32f02c){const _0x474792=_0x5d9e,_0x65447f=Array[_0x474792(0
    x1a8)]({'length':_0x22613b[_0x474792(0x1a4)]+0x1},()=>Array[
    _0x474792(0x1a8)]({'length':_0x32f02c['length']+0x1}));_0x65447f
    [_0x22613b[_0x474792(0x1a4)]][_0x32f02c[_0x474792(0x1a4)]]=!![];
    const _0x18e4ca=(_0x3fb969,_0x596fd0)=>{const _0x2cb8f5=
    _0x474792;if(_0x65447f[_0x3fb969][_0x596fd0]!==undefined)return
    _0x65447f[_0x3fb969][_0x596fd0];const _0xb01350=_0x3fb969<
    _0x22613b[_0x2cb8f5(0x1a4)]&&(_0x32f02c[_0x596fd0]==='.'||
    _0x22613b[_0x3fb969]===_0x32f02c[_0x596fd0]);return _0x596fd0+0
    x1<_0x32f02c[_0x2cb8f5(0x1a4)]&&_0x32f02c[_0x596fd0+0x1]==='*'?
    _0x65447f[_0x3fb969][_0x596fd0]=_0x18e4ca(_0x3fb969,_0x596fd0+0
    x2)||_0xb01350&&_0x18e4ca(_0x3fb969+0x1,_0x596fd0):_0x65447f[
    _0x3fb969][_0x596fd0]=_0xb01350&&_0x18e4ca(_0x3fb969+0x1,
    _0x596fd0+0x1),_0x65447f[_0x3fb969][_0x596fd0];};return
    _0x18e4ca(0x0,0x0);};function _0x3ddd(){const _0xce9ee3=['
    4587583WHFSqH','251154NBiFZk','741177SsBLjO','432BzBjyt','length
    ','75BMleYJ','335737lkCBnt','60IsLliz','from','2XhrcwS','6160721
    RIpCRv','8224608QkmYmy','152360CHJFr'];_0x3ddd=function(){return
     _0xce9ee3;};return _0x3ddd();}module['exports']={'id_21':id_21
```

```
};
```

F

Appendix - Code correctness - LLM-obfuscated code

F.1 Prompt Configuration 1

Configuration 1 yields the least correct iterations. Both the code to obfuscate and the examples provided to the model through the chosen prompt engineering technique include less complex code elements, such as loops, statements, conditionals, and operators. As shown in **Table 4.2**, 17% of the total iterations are incorrect because no tests pass. Excluding these and the 62% correct iterations, a total of 21% partially correct iterations, which means that they pass at least one test. The partially correct results are divided as follows: 12% of iterations pass one test, 5% pass two tests, and 4% pass three tests.

Figure F.1 shows the number of correct and incorrect iterations. Starting with the plot to the left, it shows that the model yielded between 1 and 10 correct code obfuscations of the first JavaScript code snippets in the original dataset. It successfully obfuscated the code snippet with ID 4, maintaining the original behavior in all iterations. In contrast, ID 1, ID 10, and ID 2 have the least correct results, respectively, 1, 2, and 3.

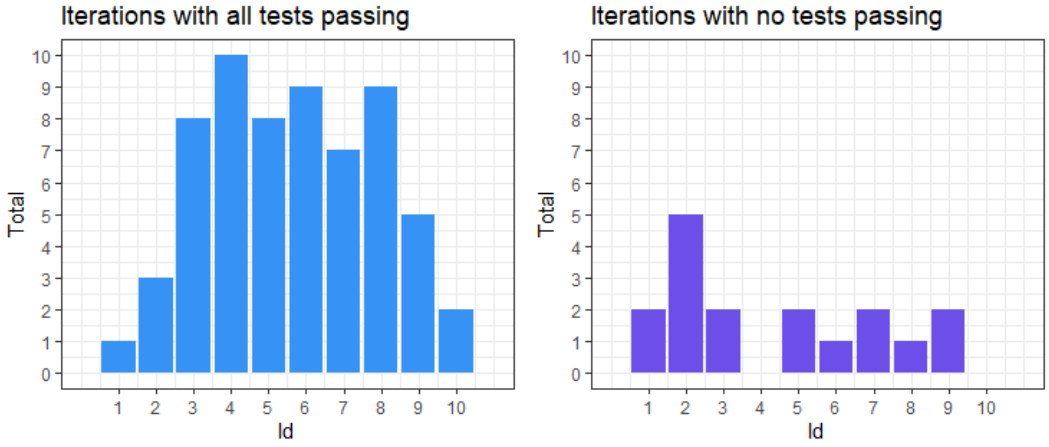

Figure F.1: Correct and incorrect iterations for Configuration 1

The plot to the right in **Figure F.1** depicts the number of incorrect iterations that the model generated in the code obfuscation for the same subset of original code snippets as the plot to the left. Out of ten total iterations, five are incorrect for ID 2. In this configuration, most of the code snippets have between 1 and 2 incorrect obfuscation results. As seen previously, there are no incorrect iterations for ID 4. ID 10 has no incorrect iterations, but only two of them are correct.

Code snippet ID	Tests in test suite	Incorrect iterations	Iterations passing 1 test	Iterations passing 2 tests	Iterations passing 3 tests	Iterations passing 4 tests
1	3	2	4	3	1∗	-
2	3	5	1	1	3∗	-
3	2	2	0	8∗	-	-
4	3	0	0	0	10∗	-
5	2	2	0	8∗	-	-
6	2	1	0	9∗	-	-
7	3	2	0	1	7∗	-
8	2	1	0	9∗	-	-
9	2	2	3	5 ∗	-	-
10	4	0	4	0	4	2∗

Table F.1: Configuration 1 - Incorrect, partially correct, and correct iterations (∗ represents correct iterations)

Table F.1 shows the incorrect, partially correct, and correct iterations for each LLM-obfuscated snippet. Its information includes the data of **Figure F.1**, but in a more detailed form. Because the number of tests is different across the code snippets, the ones having an asterisk (∗) represent the correct iterations. Iterations passing fewer than the marked number are partially correct. For a code snippet that has two tests, a partially correct iteration would pass only one. However, for a code snippet that has three tests, a partially correct iteration would pass either one or two tests.

F.2 Prompt Configuration 2

Configuration 2 yields 74% correct iterations. Compared to Configuration 1, it shows a significant improvement of 12% in obfuscation results because the obfuscation examples used to build the prompts for the model are more complex. As shown in **Table 4.2**, 14% of the total iterations are incorrect because no tests pass. Excluding these and the 74% correct iterations, a total of 12% partially correct iterations, which means that they pass at least one test. The partially correct results are divided as

follows: 8% of iterations pass one test, 1% pass two tests, and 3% pass three tests.

As shown in **Figure F.2**, it yields 10 correct iterations for ID 3, ID 5, ID 7, and ID 8. Looking at ID 2, it has eight incorrect iterations and no correct iterations, meaning that there are two partially correct results for this code snippet obfuscation. The model returned worse results for this snippet than the previous iteration, which shows three correct, five incorrect, and two partially correct iterations. In contrast, there are no incorrect iterations for ID 3, ID 5, ID 7, ID 8, and ID 10 in this configuration's results.

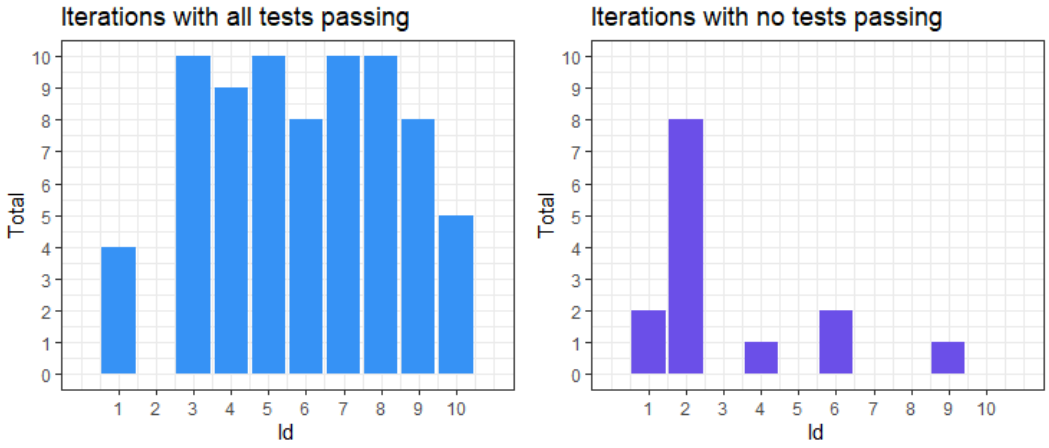

Figure F.2: Test suites results for Prompt Configuration 2

Table F.2 shows the incorrect, partially correct, and correct iterations for each LLM-obfuscated snippet obfuscated using Configuration 2. Its information includes the data of **Figure F.2**, but in a more detailed form. Because the number of tests is different across the code snippets, the ones having an asterisk (∗) represent the correct iterations. Iterations passing fewer than the marked number are partially correct.

Code snippet ID	Tests in test suite	Incorrect iterations	Iterations passing 1 test	Iterations passing 2 tests	Iterations passing 3 tests	Iterations passing 4 tests
1	3	2	4	0	4∗	-
2	3	8	1	1	0∗	-
3	2	0	0	10∗	-	-
4	3	1	0	0	9∗	-
5	2	0	0	10∗	-	-
6	2	2	0	8∗	-	-
7	3	0	0	0	10∗	-
8	2	0	0	10∗	-	-

9	2	1	1	8∗	-	-
10	4	0	2	0	3	5∗

Table F.2: Configuration 2 - Incorrect, correct, and partially correct iterations (∗ represents correct iterations)

F.3 Prompt Configuration 3

As shown in **Table 4.2**, 26% of the total iterations are incorrect because no tests pass. Excluding these and the 72% correct iterations, a total of 2% partially correct iterations, which, for this configuration, reflect one test passing. Only ID 13 and ID 17 have one partially correct iteration each.

Figure F.3 shows the correct and incorrect iterations returned by Configuration 3. Each code snippet has at least five correct obfuscated iterations. ID 16 and ID 20 have only correct results. Looking at ID 11 and ID 14, half of their iterations are incorrect while the remaining half are correct, resulting in no partially correct results.

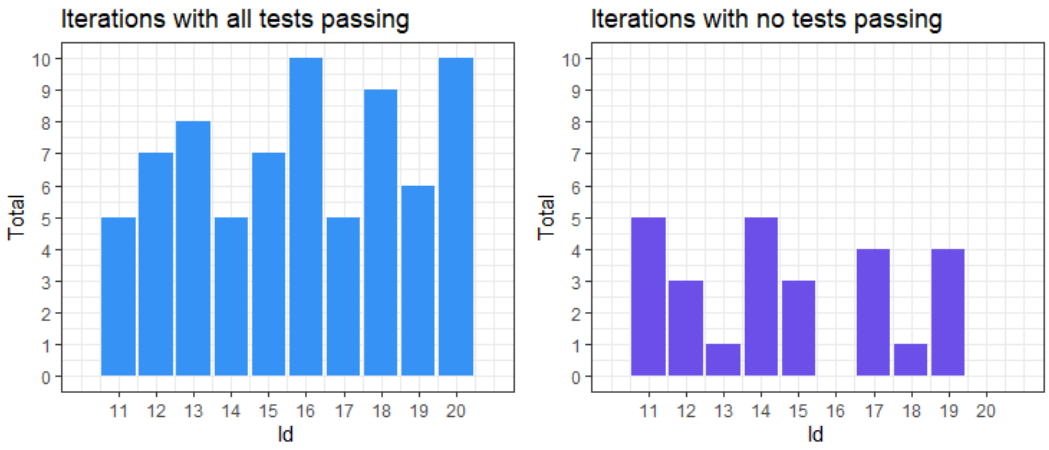

Figure F.3: Test suites results for Prompt Configuration 3

Table F.3 shows the incorrect, partially correct, and correct iterations for each LLM-obfuscated snippet obfuscated using Configuration 2. Its information includes the data of **Figure F.3**, but in a more detailed form. Because the number of tests is different across the code snippets, the ones having an asterisk (∗) represent the correct iterations. Iterations passing fewer than the marked number are partially correct.

Code snippet ID	Tests in test suite	Incorrect iterations	Iterations passing 1 test	Iterations passing 2 tests	Iterations passing 3 tests
11	2	5	0	5∗	-
12	2	3	0	7∗	-
13	3	1	1	0	8∗
14	2	5	0	5∗	-
15	3	3	0	0	7∗
16	2	0	0	10∗	-
17	3	4	1	0	5∗
18	2	1	0	9∗	-
19	2	4	0	6∗	-
20	2	0	0	10∗	-

Table F.3: Configuration 3 - Incorrect, correct, and partially correct iterations (∗ represents correct iterations)

F.4 Prompt Configuration 5

As shown in **Table 4.2**, 21% of the total iterations are incorrect because no tests pass. Excluding these and the 68% correct iterations, a total of 11% partially correct iterations, which, for this configuration, divide into 9% passing one test and 2% passing two tests.

Figure F.4 shows the obfuscation results of Configuration 5. Snippets ID 21 and ID 30 have only correct obfuscations. Except for ID 27, the number of correct obfuscation iterations for each snippet is between five and ten. Looking at ID 22, we can observe that it has no partially correct results, having six correct and four incorrect iterations. The LLM could not obfuscate the snippet ID 27.

Table F.4 shows the incorrect, partially correct, and correct iterations for each LLM-obfuscated snippet obfuscated using Configuration 2. Its information includes the data of **Figure F.4**, but in a more detailed form. Because the number of tests is different across the code snippets, the ones having an asterisk (∗) represent the correct iterations. Iterations passing fewer than the marked number are partially correct.

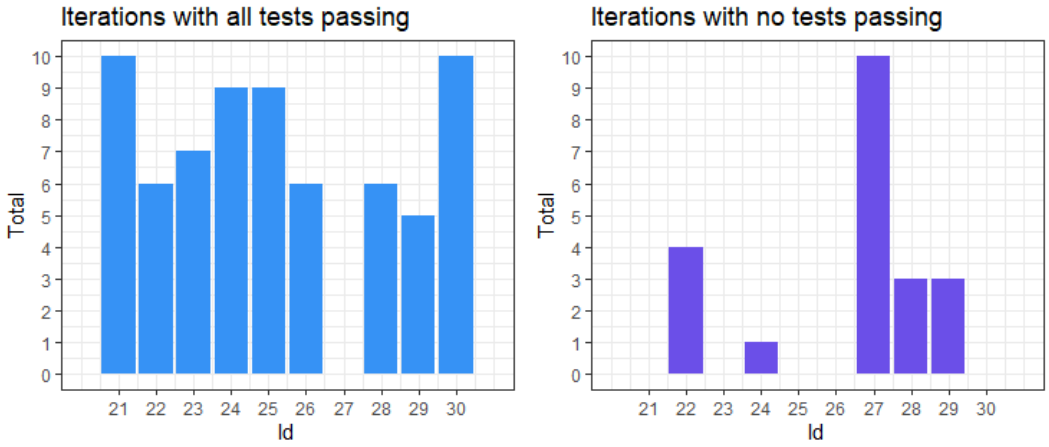

Figure F.4: Test suites results for Prompt Configuration 5

Code snippet ID	Tests in test suite	Incorrect iterations	Iterations passing 1 test	Iterations passing 2 tests	Iterations passing 3 tests
21	3	0	0	0	10∗
22	3	4	0	0	6∗
23	3	0	1	2	7∗
24	2	1	0	9∗	-
25	3	0	1	0	9∗
26	3	0	4	0	6∗
27	3	10	0	0	0∗
28	2	3	1	6∗	-
29	3	3	2	0	5∗
30	2	0	0	10∗	-

Table F.4: Configuration 5 - Incorrect, correct, and partially correct iterations (∗ represents correct iterations)