
EVALUATION OF LARGE LANGUAGE MODELS ON CODE OBFUSCATION *

Adrian Swindle
Saint Louis University
St. Louis, MO
adrian.swindle@slu.edu

Derrick McNealy
University of Southern Mississippi
Hattiesburg, MS
derrick.mcnealy@usm.edu

Giri Krishnan
University of California, San Diego
San Diego, CA
gkrishnan@ucsd.edu

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

ABSTRACT

Obfuscation intends to decrease interpretability of code. Large Language Models (LLMs) have been proposed for code synthesis and code analysis. This paper looks to understand how well LLMs can detect obfuscations in code. This is achieved by systematically evaluating several LLMs' capabilities to detect obfuscated code and provide explanation across a variety of obfuscation techniques with varying levels of complexity. We found that LLMs are better at correctly detecting obfuscations that changed identifiers, even to misleading ones, compared to obfuscations involving code insertions (unused variables, as well as variables that replace constants with expressions that evaluate to those constants). The obfuscations that layered multiple simple transformations were poorly detected by LLMs. For these, only 20-40% of the LLMs' responses were correct. We also found that adding misleading documentation was successful in misleading LLMs. Overall, our results suggest a gap in LLMs ability to understand code.

Keywords Large Language Models · Obfuscated Code · ChatGPT · Jurassic-2 · PaLM

1 Introduction

Code obfuscation is a series of transformations that maintain the functionality of a program while making it harder to understand or reverse engineer [1]. Obfuscated code remains a challenge for cybersecurity due to its ability to mask malware from conventional detection methods[1].

Large Language Models (LLMs) have been proposed for analyzing obfuscated code and are also widely used in code synthesis. Here, we study three LLMs (ChatGPT 3.5, Jurassic-2, and PaLM) with respect to their ability to detect obfuscation or code behavior. Theoretically this problem, in general, is undecidable. However, we are interested in very simple code which always terminates, whose behavior can be understood, and termination proved easily. Detecting obfuscation involves understanding code behavior and would be a specific test of reasoning for the LLMs.

1.1 Base Code

We used 21 distinct C++ base codes². All the base codes used compute simple functions, such as those that would be used in an introductory programming course. Complexity varied from printing integers 1 to 10 each in a new line, to checking whether the input is prime. Data structures and controls structures (code with and without loops and

* *Citation:* Authors. Title. Pages.... DOI:000000/11111.

²https://github.com/SwindleA/ObfuscationDatabase/tree/main/Base_Code

recursion) used were also varied to ensure that the LLMs are tested against a wide spectrum of coding tasks. We also included pieces of code whose behavior was simple but uncommon (checking whether input excluding the letter ‘x’ is a palindrome, printing a space followed by 6 newlines, etc.). These pieces of code were included to lower the likelihood of the LLMs having encountered them during training. Table 14 lists the base code and their descriptions.

1.2 Obfuscation

A code obfuscation is a series of transformations that maintain the logic of a program while making it harder to understand or reverse engineer [1]. Eighteen obfuscations were created for this project³. We used obfuscations with varying complexity. They can be grouped as (i) obfuscations that do not change the abstract syntax tree: These include transformations such as removing spaces and new lines, changing identifier names (shuffling the identifiers already used in the base code, using random identifiers, using misleading identifiers etc.), changing strings to ASCII etc. One obfuscation of note inserted misleading documentation. (ii) Obfuscations that change the abstract syntax tree: These include transformations like inserting unused variables, unnecessary statements (if-then statements, for-loops), changing math constants with complex expressions that evaluate to constants, replacing for-loops by recursion etc. (iii) Layered obfuscations that combined multiple transformations.

1.2.1 O17 and O18

O17 and O18 were created specifically to test Jurassic’s ability further. After getting initial results for the first sixteen obfuscations, Jurassic appeared to be the best at correctly identifying the code. O17’s main objective was to make the code look like its functionality was slightly different. For example, if the original code took the sum of 1-10, then the obfuscation would make the code look like it took the product of 1-10. The main goal of O18 was to create confusion and ambiguity about the code’s true purpose and operation. This was achieved by using non-descriptive variable names, introducing redundant operations, and altering the logical structure of the code. The goal was to make the code more difficult to interpret at first glance. More details on O17 and O18 in Section 2.3.

1.3 Data

1.3.1 Prompting the LLMs

The LLMs were tested using 3 prompts : (i) “Do these pieces of code achieve the same goal?” (ii) “Is the functionality of these pieces of code the same?” (iii) “What does this piece of code do?”. Prompts 1 and 2 included the obfuscated and original code. The words "goal" and "functionality" in prompts 1 and 2 respectively, are the key differences between the prompts. Prompting the LLM with "goal" was aimed at leading it towards giving an answer regarding the outcome of the code. "Functionality" was used in hopes of leading the LLM to analyze how the code functions instead of the outcome. Prompt 3 differed from 1 and 2 because it does not include the original code. The prompt is purposefully vague to see how well the LLM understands the code without any context. The prompts are shown in Table 1.

Table 1: Prompts given to the LLMs

Prompt 1	Prompt 2	Prompt 3
Do these pieces of code achieve the same goal? <i>base code</i> AND <i>obfuscated code</i>	Is the functionality of these pieces of code the same? <i>base code</i> AND <i>obfuscated code</i>	What does this piece of code do? <i>obfuscated code</i>

1.3.2 Gathering Data

Data was gathered using the free version of each API that accompanied each LLM. All the data gathered was then organized into Excel spreadsheets. Each LLM had its own workbook with two sets of sheets. One set of sheets is organized such that each sheet is named after a base code, i.e., B1, B2, etc... The basic contents can be seen in Table 2. The second set of sheets within a workbook were sheets named after the obfuscations (See Table 3).

A Python script was used to generate the spreadsheets with the code inserted into them with the correct format. Another script was then used to automatically create the prompt with the preset template and use the LLM’s API to ask feed the LLM the prompt. The prompt and response were then inserted into the appropriate places in the workbooks.

³See Table 13

Table 2: Organization of Results by Base Code

Obfuscation	Code	Prompt	Response	Correctness	Notes
O1	Obfuscated code	Prompt given to LLM	The LLM's response to the prompt	Correctness rating	Optional

Table 3: Organization of Results by Obfuscation

Base Code	Code	Prompt	Response	Correctness	Notes
B1	obfuscated code	Prompt given to LLM	The LLM's response to the prompt	Correctness rating	Optional

1.3.3 API

The API for the three LLM's is very similar. The API call for each of the LLM's is some form of ChatCompletion⁴. This ChatCompletion differs from the other API calls that could be used because it mimics the behavior of the web client chat. The specific model used for ChatGPT is 'gpt-3.5-turbo'. This means that the model used is 3.5 version of ChatGPT and it is the most up-to-date version of ChatGPT [2]. The alternatives were snapshots of the model on a specific date. The model version of Jurassic used is 'j2-ultra'. Jurassic 2 Ultra is the second version of Jurassic and ultra meaning the most powerful version of Jurassic [3]. PaLM uses 'chat-bison' as its model. Bison is the foundational model that PaLM uses [4].

1.3.4 Temperature and Top P

Two major settings for the LLM's API are temperature and top p. Temperature "control[s] the randomness and creativity of the generated text in a generative language model" [5]. In other words, this controls how much freedom the LLM has in generating a response. The range of temperature is from 0 to 2. Zero being a extremely concise answer and two being highly creative. The temperature used for this project is 0.7. This temperature is used because ChatGPT's online chat is 0.7, which was found to be the general standard temperature for most LLMs [6].

Top p is the "threshold [that] represents the proportion of the probability distribution to consider for the next word" [7]. Top p determines the range that the LLM will choose a word from. For example, if top p is set to 0.05, then only the top 5% of words will be chosen from. This is very simplistic explanation but show the general idea of top p⁵. The top p chosen for this project is 1, giving the LLM all possible words to choose from.

1.3.5 Rating Correctness

A rating was created and used to rate how correct each LLM's response was to each prompt it was given. The correctness rating of a response fell within a spectrum that ranged from 'High Correct' to 'High Incorrect'.

Table 4: Correctness Range

Correctness
High Correct
Medium Correct
Low Correct
High Maybe
Medium Maybe
Low Maybe
Low Incorrect
Medium Incorrect
High Incorrect
N/A

⁴ChatGPT uses ChatCompletion [2]. Jurassic uses Completion [3]. PaLM uses chat [4].

⁵Top p actually uses tokens as what it selects from to generate responses. Tokens are substituted for words to help simplify the topic. See [7] for a more in depth explanation

At the top of the spectrum is High Correct. A response with a High Correct rating means that the response that the LLM gave was 99% correct and it properly explained the function of the code. A Low Correct response is one that was correct but could have contained more detail in how the code works.

The middle of the spectrum is the Maybe area. The Maybe range of correctness is a very grey area. A maybe rating represents answers that may be correct or contain a large amount of correct information. A response warrants a High Maybe if the LLM's yes/no response is incorrect, but its explanation of the code is extremely correct. Low Maybe and Low Incorrect had a lot of overlap. A Low Maybe is for if a response is almost incorrect, but enough information is accurate that it is not quite incorrect.

Low Incorrect is the slight push into incorrect from Low Maybe. High Incorrect is the case when the LLM's response and explanation are nowhere near correct. The LLM may have detailed a completely different functionality.

The rating of N/A was for the most part used in instances where an obfuscation did not apply to the base code. There were other cases where the API returned an error for prompts, so these cases were also marked as N/A.

2 Results

2.1 Overall Results

A flaw of LLMs is that they are not able to run the code. So, aspects of code such as mathematical computation or the actual coding aspects are all understood purely through what the LLM has seen before. Obfuscation 9 specifically targets math, but there are also instances where the LLM can get non-computational aspects incorrect. Table 5 is an example as shown by asking Jurassic prompt 1 with O1 applied to B5.

Table 5: Jurassic Concatenating Incorrectly

Base Code	Response
<pre> 1: #include <iostream> 2: int main(){ 3: std::string firstHalf = "race"; 4: std::string secondHalf = "car"; 5: std::string fullWord = firstHalf.append(secondHalf); 6: std::cout<<fullWord<<std::endl; 7: }</pre>	<p>Yes, both code snippets achieve the same goal, which is to concatenate two strings and store the result in a new string. In the first code snippet, the secondHalf string is concatenated with the string returned by <code>firstHalf.append(secondHalf)</code>, while in the second one, the secondHalf string is concatenated with the string returned by <code>firstHalf.append(secondHalf)</code>. Both code snippets result in the same output, which is "carrace".</p>
Obfuscated Code	Correctness Rating
<pre> 1: #include <iostream> 2: int main(){ 3: std::string secondHalf = "race"; 4: std::string firstHalf = "car"; 5: std::string fullWord = secondHalf.append(firstHalf); 6: std::cout<<fullWord<<std::endl; 7: }</pre>	<p>Medium Maybe</p>

Obfuscation 1 is the simplest obfuscation. The variables 'firstHalf' and 'secondHalf' are switched. In this instance the 'first code snippet' refers to the obfuscated code. The descriptions, as shown in Table 5, of both codes say that the variable 'secondHalf' is appended to the variable 'firstHalf'. This is incorrect and the output that Jurassic gives is the opposite of the desired result. This is just a microcosm of where generally the LLMs fell short. Obfuscation is successful because LLMs do not run the code and they can make simple mistakes.

2.2 ChatGPT

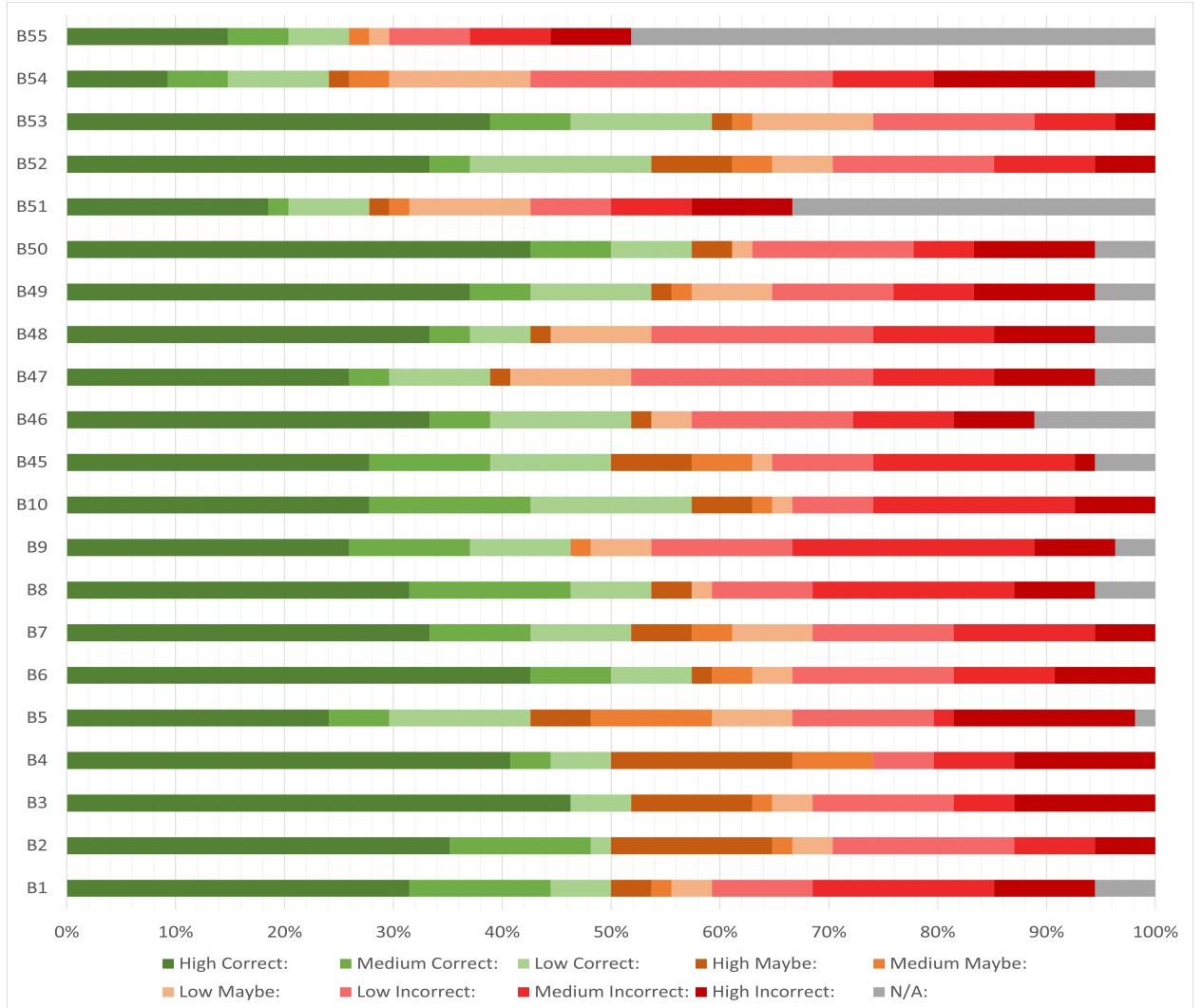
ChatGPT did not perform nearly as well as Jurassic and preformed similarly to PaLM. One interesting feature of ChatGPT's assessment of the code was that it would often deem the goal of the obfuscated code different from the base code but would then explain itself profusely. In its explanation, it would outline why the goal of both codes were in fact the same, regardless of it previously saying the opposite incorrectly.

Table 6: ChatGPT Contradiction

Base Code	Correctness Rating
<pre> 1: #include <iostream> 2: int main(){ 3: std::string hello = "hello"; 4: for(int i =0; i<10; i++){ 5: std::cout<<hello; 6: } 7: }</pre>	High Incorrect
Obfuscated Code	Response
<pre> 1: #include <iostream> 2: #include <algorithm> 3: #include <sstream> 4: int main() { 5: std::string hello = "hello"; 6: std::ostringstream oss; 7: for (int i = 0; i < 10; i++) { 8: oss << hello; 9: } 10: std::string output = oss.str(); 11: // Reverse the output string 12: std::reverse(output.begin(), output.end()); 13: // Enclose the output string in square brackets 14: output = "[" + output + "]"; 15: std::cout << output << std::endl; 16: }</pre>	<p>No, the two pieces of code do not achieve the same goal.</p> <p>The first piece of code simply prints the string "hello" 10 times using a for loop and the 'std::cout' statement.</p> <p>The second piece of code does the following:</p> <ol style="list-style-type: none"> 1. It uses an 'std::ostringstream' object 'oss' to concatenate the string "hello" 10 times using a for loop and the 'oss << hello' statement. 2. It converts the contents of 'oss' to a 'std::string' using the 'oss.str()' function and assigns it. 3. It reverses the characters in the 'output' string using the 'std::reverse' function. 4. It encloses the 'output' string in square brackets by concatenating "[" and "]" to it. 5. It prints the modified 'output' string using the 'std::cout' statement. <p>So, the second piece of code not only prints the string "hello" 10 times, but also modifies and manipulates the output before printing it.</p>

In this example shown in Table 6, B6, printing "hello" ten times, is applied with O16. In terms of obfuscation, the second piece of code incorporates additional steps to alter the repeated string "hello". Firstly, O16 uses an ostream to build the repeated string, which confuses the LLM. Secondly, it reverses the repeated string, which misleads the LLM into thinking it's producing a different string than just "hello" repeated. Lastly, the addition of square brackets at the beginning and end further disguises the actual functionality of the code. The response from ChatGPT is noteworthy because it stated in the response that the code prints hello ten times but goes on to contradict itself saying that both codes do not have the same goal.

Figure 1: ChatGPT: Base Code

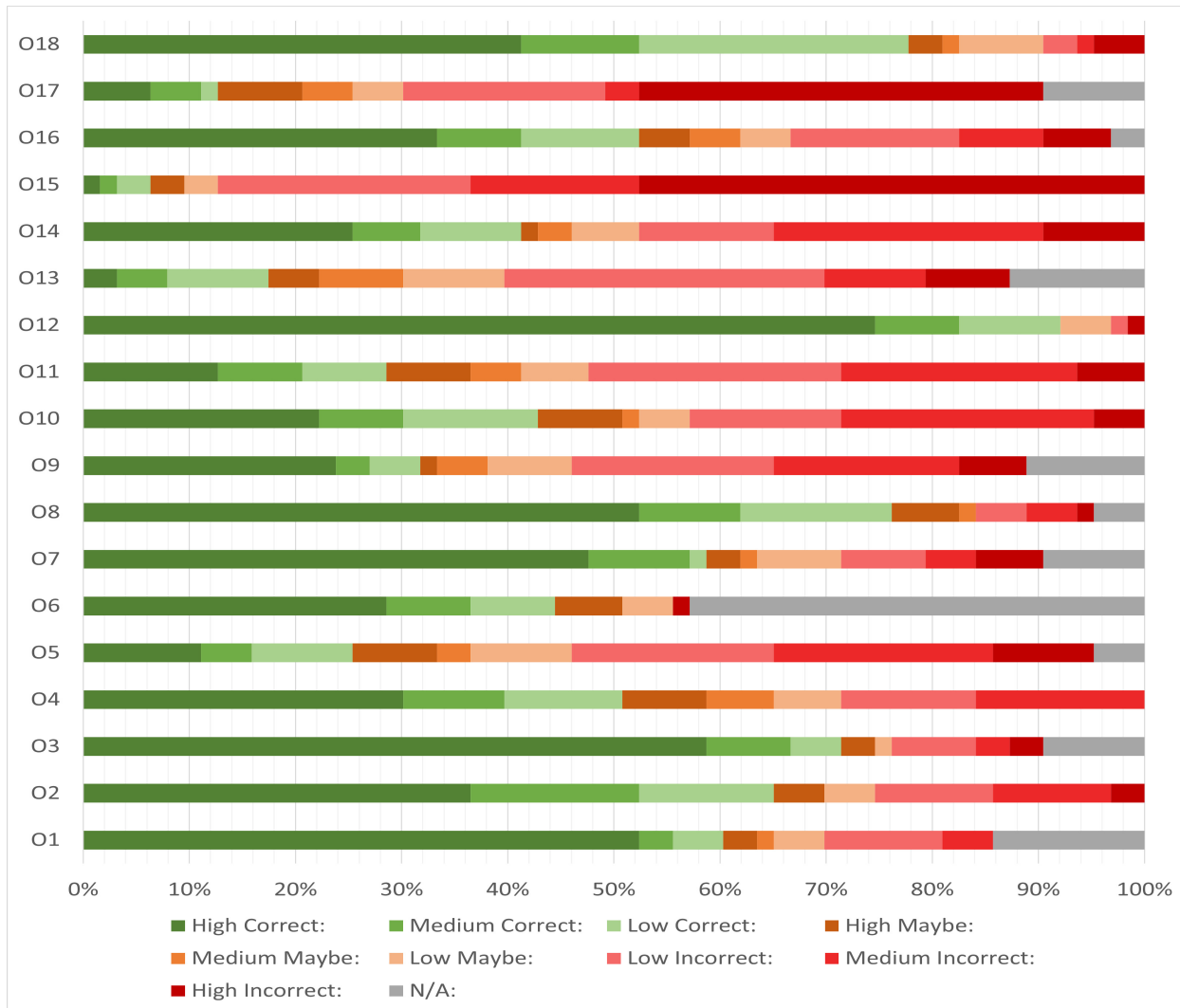


2.3 Jurassic

Jurassic was by far the best at identifying obfuscated code with 683 correct responses out of 1053 prompts given. Even though its results were great, Jurassic still occasionally had its hiccups. Before going into specific aspects that Jurassic got incorrect, Table 7 shows an outlier case where Jurassic contradicted itself. This is O6 applied to B9 asking Q1.

Base code 9 prints the odd numbers in the given vector. Obfuscation 6 changes the code to use recursion instead of a for-loop. If the answer needed from the LLM was yes or no, then the answer would be correct. The issue is that the description of the code is contradictory. In Table 8, the second sentence of the response says that the code will "print only the even numbers to the console". The very next sentence, which describes the recursive function, says that odd numbers are printed. The existence of self-contradiction is an issue, but it would have made more sense if

Figure 2: ChatGPT: Obfuscation



the obfuscation involve the word 'even' to imply even numbers are wanted. This type of contradiction did not happen enough to be a major issue but even one case should be notable.

After running the initial 16 obfuscations, it was clear that Jurassic was the best LLM. To get more instances of incorrect responses from Jurassic, O17 and O18 were created to focus on Jurassic's weaknesses and exploit them.

2.3.1 O17

The general goal of O17 was to create an obfuscation that kept the code clean and focus on Jurassic's weaknesses. Keeping the code clean means that O17 does not contain the features that O15 had that made the O15 codes look like a jumbled mess. Two features of the first 16 obfuscations' results to note is the high rate of incorrectness in O2 and B3. Obfuscation 2 focused on comments. This shows that Jurassic, like all LLMs, uses the comments to evaluate the code's functionality. The other notable thing is the high rate of incorrect answers that B3 caused. Base code 3 is a simple program that takes the product of all the numbers 1-10. The use of the variable 'sum' to keep the total number caused lots of confusion for Jurassic and the other LLMs (more on B3 in section 2.5). With these features in mind, O17 aimed at using comments and variables to mimic a behavior similar in structure but completely different in functionality.

Table 9, shows obfuscation 17 applied to base code 1. Base code 1 prints the numbers 1-10. The obfuscation aims at pretending to find the sum of the numbers 1-10. The comments explain the functionality as this. The comment "Ignore

Table 7: Jurassic Contradicting itself (Part 1)

Base Code	Obfuscated Code
<pre> 1: #include <iostream> 2: #include <vector> 3: int main(){ 4: std::vector<int> numbers = {1,2,3,5,2,1345, 63,56,3,6626, 2556,342,5345,55,44,33,223,863, 24,52,98,888}; 5: for(int i =0; i<numbers.size(); i++){ 6: if(numbers[i] %2 != 0){ 7: std::cout<<numbers[i] <<" "; 8: } 9: } 10: }</pre>	<pre> 1: #include <iostream> 2: #include <vector> 3: void printOdds(const std::vector<int> & numbers, int index) { 4: if(index < numbers.size()) { 5: if(numbers[index] % 2 != 0) { 6: std::cout << numbers[index] << " "; 7: } 8: printOdds(numbers, index + 1); 9: } 10: } 11: int main() { 12: std::vector<int> numbers = {1,2,3,5,2,1345,63,56,3,6626, 2556,342,5345,55,44,33,223,863,24, 52,98,888}; 13: printOdds(numbers, 0); 14: return 0; 15: }</pre>

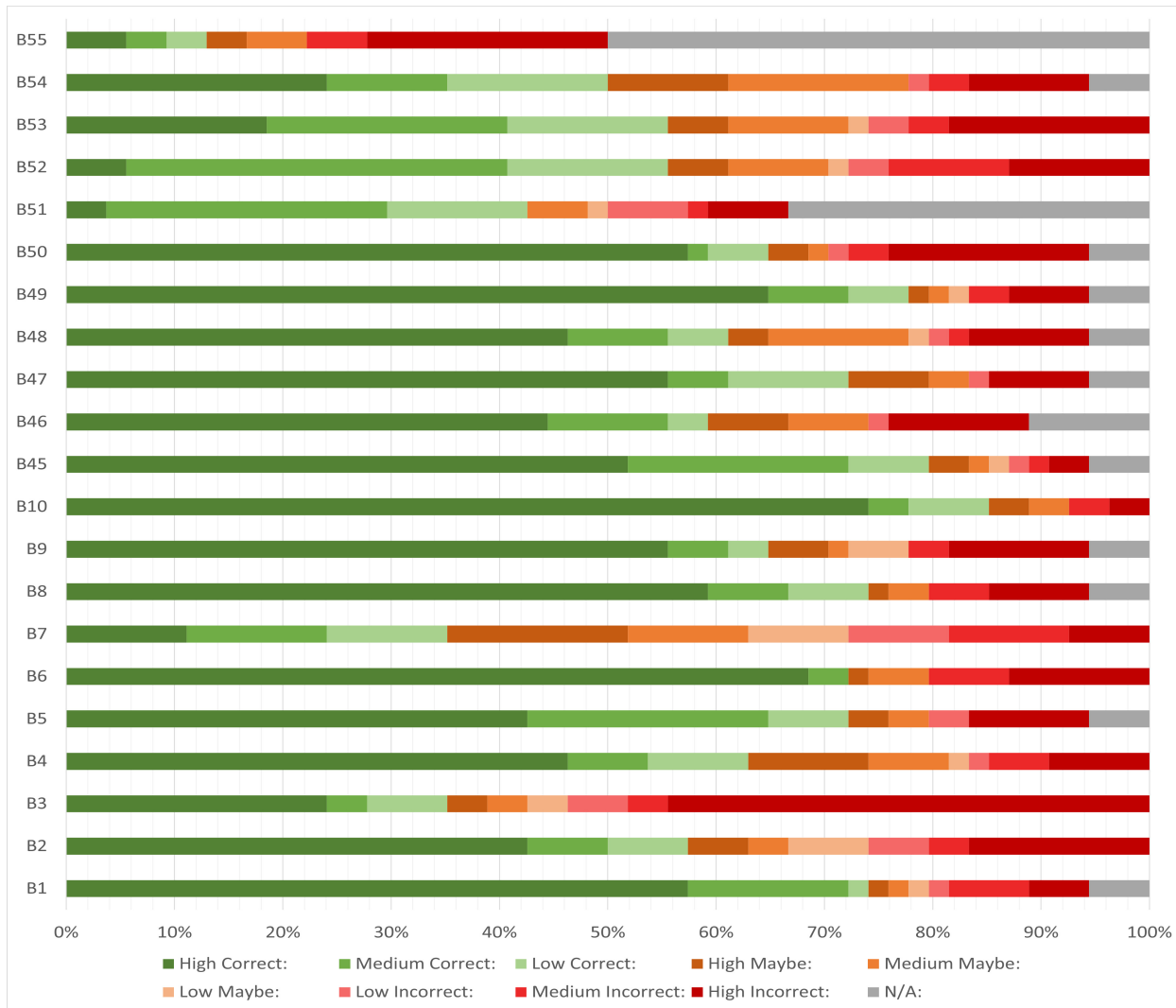
Table 8: Jurassic Contradicting itself (Part 2)

Truncated Response
Yes, both pieces of code achieve the same goal. They take an input vector of integers, and print only the even numbers to the console. The printOdds function in the first code sample is a recursive function that iterates through the input vector and prints each odd number to the console, while the for loop in the second code sample iterates through the input vector and prints each odd number to the console.
Correctness Rating
Low Maybe

this, this is for testing" was especially relevant for prompt 3, which simply asks the LLM what the obfuscated code does without giving it the base code. ChatGPT went as far as saying "[t]he code also includes some additional lines for testing purposes, but these do not affect the main functionality of the program". By having an untrue comment, the LLM simply ignores that portion of the code. The second piece of the obfuscation is the variable names. The first part of the obfuscated code uses a variable 'sum' to find the sum of 1-10. What truly confused the LLMs is the line that prints the character/string 'sum'. This throws warnings to the compiler but still runs and prints the decimal value of the string 'sum'. The LLMs saw this as printing the variable 'sum' instead of the string sum. Jurassic's response to asking prompt 3 with the obfuscated code shown in Table 9 specifically says that the "value of 'sum'" is printed. The 'sum' is in reference to the variable not the string.

For Jurassic the results from O17 were much better than the other obfuscations except O15. O15 had 47 incorrect responses while O17 only had 31. Even though O17 did not surpass O15 in tricking Jurassic, its results are much better than the other obfuscations and the code was clean. More on O17's overall results in section 2.6

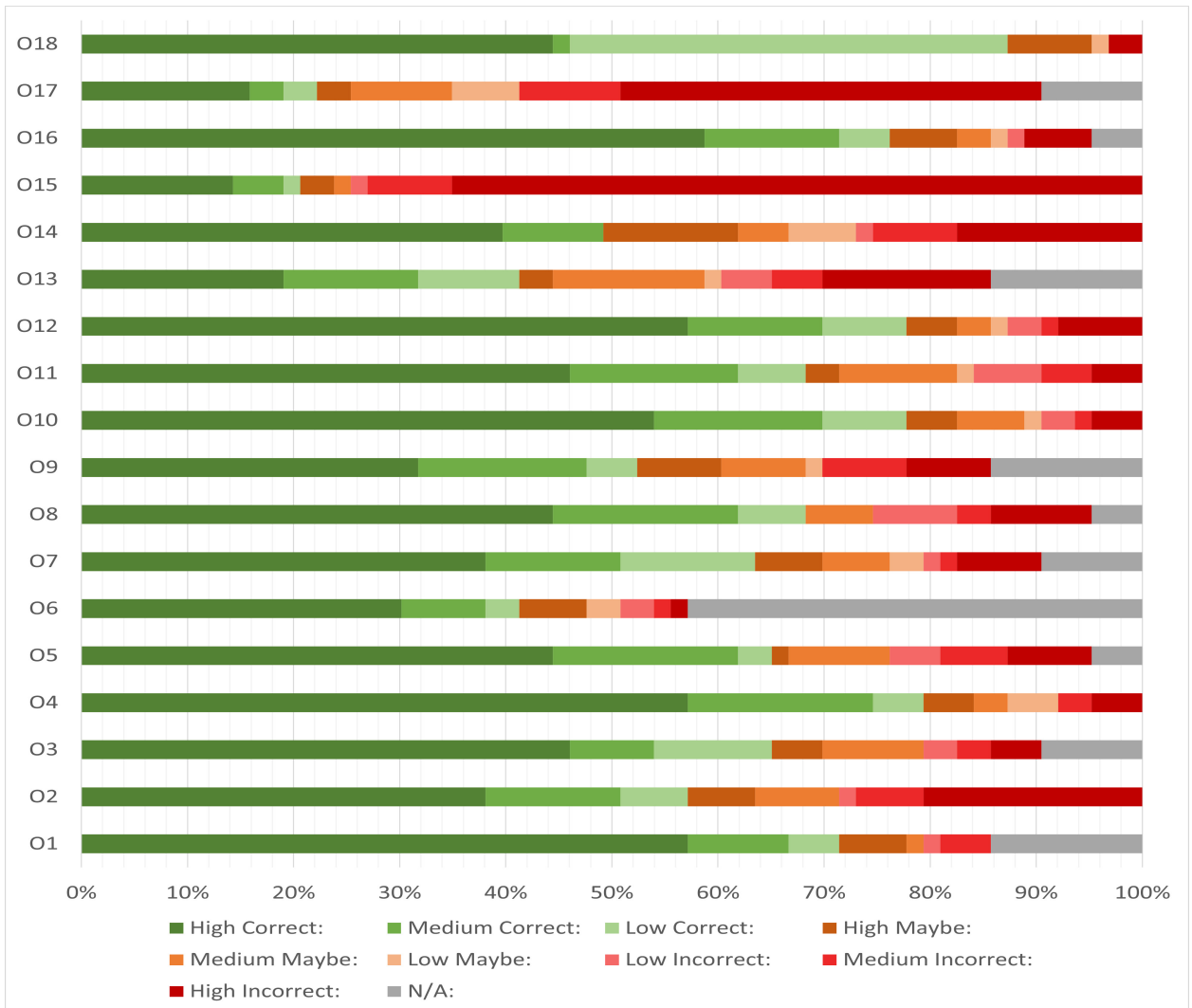
Figure 3: Jurassic: Base Code



2.3.2 O18

O18 was created in response to the high rate of incorrectness regarding Jurassic's ability to determine the functionality of obfuscated code. This rate of incorrectness was seen the most in the application of O15 to the base code. O18 was an attempt to recreate that effect, but through methods that were less abstract and overwhelming. It contains techniques that can be seen throughout previous obfuscations such as variable renaming and misleading operations. Its main focus was to use a series of commenting for each obfuscation that precisely described the base code that was being obfuscated. This was in hopes of tricking Jurassic into thinking both code functionalities were the same. Table 10, shows obfuscation 18 applied to base code 52. The obfuscation here effectively presents the code as a complex string manipulator while the base functionality is merely printing out a string of randomly generated characters separated by '!'. The intricate use of separate loops, the introduction of a seemingly significant '!' character, and the utilization of random number generation and ASCII conversions all contribute to the robustness of the obfuscation. Furthermore, the two separate loops used to print the random characters before and after the '!' add a layer of complexity to the obfuscation. Jurassic considers these loops as working in tandem or having interdependent functionalities, while in reality, they are independent, each printing a separate set of random characters. This successfully throws off Jurassic's comprehension of the code.

Figure 4: Jurassic: Obfuscation



For Jurassic the results from O18 were much worse than the others and overall did not do a good job at replicating the results O15. In this isolated instance, Jurassic was confused, but overall, O18's mix of techniques proved to be the worst at tricking Jurassic. More on O18's overall results in the section on the obfuscations.

2.4 PaLM

There is an odd quirk of PaLM's that happens in instances that the obfuscation included ASCII characters. There were multiple instances that it would say the ASCII characters will output some variation of 'Hello World'. Below is an example response to prompt 1 for obfuscation 13 applied to base code 54.

There are two main obfuscations applied to this code. The first is changing the string 'Input', which is printed to the output, to its ASCII equivalent. The second is changing the strings '>' and '2' to their ASCII. This script is supposed to take a word as input and replace the middle character with '>' and check if the second character is equal to '2'. The PaLM focused on the simple input without considering what the actual output is. It is also notable that PaLM got the base code description incorrect also. In most cases, the LMs will get the base code correct regardless of the complexity of the obfuscation. It appears in this instance, PaLM decided that the section of the obfuscated code that is the series of for-loops is the most important aspect of either piece of code. Because PaLM focused on this one section, its misinterpretation spreads as the overarching description of both the obfuscated and base code. The misinterpretation of the ASCII characters to be 'Hello World' is lent to the probability aspect of the LLM. Anyone who has done beginner

Table 9: O17 Example: Base Code 1

Base Code	Obfuscated Code
<pre> 1: #include <iostream> 2: int main(){ 3: for(int i =1; i<=10;i++){ 4: std::cout<< i << std::endl; 5: } 6: }</pre>	<pre> 1: //This program finds the sum of the nums from 1 to 10 2: #include <iostream> 3: int main(){ 4: //Find the sum of the numbers 5: int sum =0; 6: for(int i =1; i<=10;i++){ 7: sum += i; 8: } 9: //Print the sum 10: std::cout<<"Sum: "<<'sum' <<std::endl; 11: //Ignore this, this is for testing 12: std::cout<<"The following is for testing only."<<std::endl; 13: for(int i = 0; i<10; i++){ 14: std::cout <<"{" <<sum-(i+45)<<"}" <<std::endl; 15: } 16: /// 17: }</pre>

level coding has done a "Hello World" project. The abundance of instances of "Hello World" appearing in PaLM's data makes it understandable on why, when it cannot figure out what the ASCII characters are, it defaults to 'Hello World'.

Figure 5 shows how well PaLM did per base code. It follows the trend of struggling with B3. The notable thing for PaLM is that there is a higher amount of N/A. PaLM had the most instances of the API returning an error when given the prompt⁶.

The previously mentioned errors that PaLM returns has potential to obscure how well PaLM did. Just for obfuscation 6, there is a 20% increase in 'N/A' responses from Jurassic to PaLM. It is not likely that PaLM would surpass Jurassic's results, but as shown later in Section 2.8, ChatGPT and PaLM are very close in effectiveness. The 57 extra prompts that ChatGPT was able to answer, might have been greatly in ChatGPT's favor.

2.5 Base Code

Prior to gathering results, it was predicted that the B45-55 will cause more incorrect responses than B1-10. This was not true. B45-55 were intended to have more complex functionality. There is an even mixture of correct and incorrect responses, but the majority of correct responses lie within B45-55. as shown in Figure 7. This is due to the use of functions and more variables. The LLMs rely on variables and function names to understand the function of the code. B45-55 utilized functions more than B1-10.

Another interesting aspect to note is the high number of incorrect answers B3 caused. It is not surprising that B55 caused the most incorrect answers because of the vague nature of the code. B3 is quite the opposite of B55. B3 calculates the product of the numbers 1-10. The common answer to a prompt involving B3 was that the code calculated the sum of the numbers 1-10 (which is B2). The differences in taking the sum and product are only the mathematical symbols and the start value of 'sum', which held the product of the numbers. When looking at the responses, it appears that the LLMs ignored the '*' and only looked at the variable 'sum'. It seems that the use of sum in the base code caused an accidental obfuscation in the base code, which in turn caused incorrect descriptions of the base code by the

⁶Time constraints prevented a proper investigation into why.

Table 10: O18 Example: Base Code 52 (truncated)

Base Code	Obfuscated Code(truncated)
1: #include <cstdlib>	1: // Include standard input/output stream library
2: #include <iostream>	2: #include <iostream>
3: #include <time.h>	3: // Include string library to work with string objects
4: using namespace std;	4: #include <string> // <-- this is the correct include for std::string
5: int main()	5: // Use standard namespace to avoid prefixing every standard library class or object with "std::"
6: {	6: bool isPalindromeWithoutX (const std::string& word) {
7: srand(time(0));	7: // Declare the start and end variables
8: for(int outerloop = 4;	8: int start = 0;
9: outerloop>0; outerloop--){	9: int end = word.size() - 1;
10: for(int innerloop1 =	10: // The main function, where the execution of the program starts
11: rand()%10;	11: while (start < end) {
12: innerloop1 >0; innerloop1--){	12: // Declare a string variable to hold the user input
13: char firstHalf=	13: if (word[start] == 'x') {
14: 'a' + rand()%26;	14: start++;
15: cout<<firstHalf;	15: // Prompt the user to enter input
16: }	16: continue;
17: cout<<"!";	17: // Read the user's input into the "word" string
18: for(int innerloop2 =	18: }
19: rand()%10; innerloop2 >0;	
20: innerloop2--){	
21: char secondHalf=	
22: 'a' + rand()%26;	
23: cout<<secondHalf;	
24: }	
25: cout<<endl;	
26: }	
27: return 0;	
28: }	

LLMs. The similarity in finding the sum of numbers and product of numbers caused confusion for the LLM on the function of the base code which translated to their understanding of the obfuscated code.

2.6 Obfuscation

The results for how well each obfuscation confused the LLMs is not entirely surprising but there are a few outliers.

O15 and O17 were the two best obfuscations by a wide margin. O17 was nearly 20% better than the 3rd best obfuscation O13. O15 was 15% better than O17. There is a nearly 70% gap between O15 and the worst obfuscation, O18. It is no surprise O15 is the best obfuscation due to the complexity of the obfuscation. It combined nearly every obfuscation technique used in O1-O14. Where O17 could potentially be better than O15 is the cleanliness of the code. O15 relied heavily on overwhelming and confusing the LLM, but in theory, a trained LLM could see that the code is obfuscated. There were instances of the LLM saying that the O15 code is potentially obfuscated code. O17 does not appear to be obfuscated. As previously mentioned, O17's goal was to zone in on Jurassic's weaknesses and keep the code as clean as possible. At a glance, a human could not tell O17 is obfuscated, but a human could easily tell that something is wrong about O15 code.

A notable obfuscation is O2. Obfuscation 2 changes nothing in the code except adding comments. O2's success rate in confusing the LLMs shows the reliance that the LLMs have on comments.

It was noteworthy how much O18 failed at tripping up Jurassic. It's especially interesting because O18 was created with the expressed intention of attacking Jurassic's weakness. One of the possible reasons for this result is Jurassic's utilization of the inaccurate comments, which detailed the functionality of the base code, throughout the obfuscated

Table 11: PaLM Defaulting to 'Hello Wold' (Part 1)

Base Code	Obfuscated Code
<pre> 1: #include <iostream> 2: #include <string.h> 3: using namespace std; 4: int main(){ 5: string *thing; 6: string word; 7: cout<<"Input: "<<endl; 8: cin>>word; 9: thing = &word; 10: word[word.size()/2] = '>'; 11: word[word[1]] == '2'; 12: cout<<*thing<<endl; 13: return 0; 14: }</pre>	<pre> 1: #include <iostream> 2: #include <string.h> 3: using namespace std; 4: int main(){ 5: string *thing; 6: string word; 7: for(int j = 1; j>0;j--) {cout<<static_cast<char>(72+j);} 8: for(int j = 1; j>0;j--) {cout<<static_cast<char>(109+j);} 9: for(int j = 1; j>0;j--) {cout<<static_cast<char>(111+j);} 10: for(int j = 1; j>0;j--) {cout<<static_cast<char>(116+j);} 11: for(int j = 1; j>0;j--) {cout<<static_cast<char>(115+j);} 12: for(int j = 1; j>0;j--) {cout<<static_cast<char>(57+j);} 13: for(int j = 1; j>0;j--) {cout<<static_cast<char>(31+j);} 14: cout<<endl; 15: cin>>word; 16: thing = &word; 17: word[word.size()/2] = 18: static_cast<char>(62*(2/9)+27*2+8); 19: word[word[1]] == 20: static_cast<char>(72-(2/9)-22); 21: cout<<*thing<<endl; 22: return 0; 23: }</pre>

code. It is possible that Jurassic used these comments to understand that the two codes had similar functionalities, disregarding the obfuscated code itself.

2.7 Prompts

Figure 9 shows the results for each prompt. Across the board, the responses to the prompts were about the same. Prompt 2 (Q2) caused the most incorrect answers at 361 but 113 of them were low incorrect. The high amount of low incorrect and the amount of 'Maybe' being like the other prompts show that Prompt 2 caused confusion in the incorrect direction. Prompt 3 (Q3) is on the opposite end. Q3 had the highest response rate in the 'Maybe' category and had the highest number of 'Low Correct' responses. This shows that Q3 incurred a high amount of mostly correct answers and Q2 incurred a high amount of partially correct answers. Another notable observation is that the results comparing Q1/2 versus Q3 show that the inclusion of the base code with the obfuscation does not significantly alter the answers to the prompts. It could be argued that the inclusion of the base code with the obfuscation hinders the LLM's ability to interpret the code.

Future work should include leading prompts to see if the LLMs' success rate is much higher if they are told that the code is obfuscated. The leading prompts can also include a hint to the functionality of the code. The prompts asked in this project all had a similar level of specificity, so it is reasonable to predict that more specific prompts should lead to better results.

Table 12: PaLM Defaulting to 'Hello Wold' (Part 2)

Truncated Response
<p>No, the two pieces of code do not achieve the same goal. The first piece of code prints the string "HELLO WORLD!", while the second piece of code prints the string "HELLO\OWORLD!".</p> <p>The difference is that the second piece of code adds a null character at the end of the string. This is because the statement 'word[word[1]] == '2'' sets the character at index 2 of the string to '2'. However, the index 2 of the string is actually the index of the null character, so setting it to '2' effectively appends the character '2' to the string.....</p>
Correctness Rating
High Incorrect

2.8 Summary of Results

Figure 10 shows how each LLM did. Jurassic is the clear winner with around 60% of its responses being correct. ChatGPT and PaLM had a similar rate of correct responses, but ChatGPT had a much lower rate of 'High Incorrect' which gives it the edge over PaLM. It is currently unclear why one LLM is better than the other. This would require research into how each LLM specifically functions. What can be understood from the results is that Jurassic and PaLM rely heavily on the variable names to identify the code. This enabled them to ignore obfuscations that included complex math. ChatGPT would get caught up in what the numbers represented, causing it to misinterpret the code. Jurassic's usage of variable names was not as much of a crutch as it was for PaLM, so instances where PaLM would only look at the variables and fail, Jurassic could ignore the variable names and see the big picture. ChatGPT's strength over PaLM is that its non-reliance on variable names led to it having the least amount of 'High Incorrect' responses out of all three LLMs.

3 Conclusion

The results shown here showcase the potential that LLMs have to help detect obfuscated code, but it also highlights the issues that need to be addressed. Simple things like variable names and comments throw the LLMs accuracy off. Jurassic has the potential to be trained to identify the obfuscations shown here and better its ability to be correct.

There are two concerns that should be identified about this research. The first concern is the high rate of answers not asked by PaLM. The number of prompts given to PaLM were lower than the other two, so those unanswered prompts could have been the difference between PaLM and ChatGPT. There is also a potential flaw in the correctness rating with the human difference of judgement. Two people rated the correctness for the responses which is two different perspectives. There is a general description for what constitutes each rating but there is still lots of room for subjectivity. To better rank the LLMs and obfuscations, a more objective method needs to be created to rate the LLMs responses.

Future directions for this research should have different focuses with the end goal of having a LLM that can identify obfuscate malware. Research with focus on the prompts will need to have prompts that are more specific to the problem. Leading prompts that hint at the function of the code or that the code is obfuscated. The use of compiled code instead of C++ can help push the LLMs ability to understand a range on programming languages. Compiled code is also harder to follow and understand than the normal programming languages. Using obfuscation software. In this project, all obfuscation was done by hand. Obfuscation software will allow the amount of data gathered to expand but also better the obfuscations done. Changing the base code to actual malware will help understand the features of malware that the LLMs struggle to understand. Finally, once the previous work has been finished, research can be done to train an LLM to identify obfuscated malware.

- [1] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Arun Kumar Sangaiah, and Aniello Cimitile. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119, 2018.
- [2] OpenAI. Gpt-3.5. <https://platform.openai.com/docs/models/gpt-3-5>, 2023. Accessed: 2023-07-14.
- [3] AI21. Jurassic-2 models. <https://docs.ai21.com/docs/jurassic-2-models#jurassic-2-ultra-unmatched-quality>, 2023. Accessed: 2023-07-14.
- [4] Google. Api documentation. <https://developers.google.com/api>, 2023. Accessed: 2023-07-14.
- [5] Lazy Programmer. What is temperature in nlp / llms? <https://medium.com/@lazyprogrammerofficial/what-is-temperature-in-nlp-llms-aa2a7212e687>, May 2023.
- [6] David Berkowitz. Temperature check: A guide to the best chatgpt feature you're (probably) not using. <https://www.linkedin.com/pulse/temperature-check-guide-best-chatgpt-feature-youre-using-berkowitz/>, Apr 2023.
- [7] Hanane D. Temperature and top p in chatgpt. "https://medium.com/@basics.machinelearning/temperature-and-top-p-in-chatgpt-9ead9345a901#id_token=eyJhbGciOiJIUzU1NiIsImtpZCI6IjY3NmRhOWQzMjJmZmlhNDI5OTMyZjU0M2U2YzFiNmU2NTEyZTQ5ODMiLCJ0eXAiOiJKV1QiFQ.eyJpc3MiOiJodHRwczovL2FjY291bnRzLmdvb2dsZS5jb20iLCJuYmYiOiJlZDQk4NzA3MzQsImF1ZCI6IjI5NjAzNTgzNC1rMwVsWGLSrQGTcsKlVCyD8uBCGqB6Th3-R6IePX1T3C9IwvGKSRdTLr42s3UogcEwkafJgPC9dtHINPe66xo0Unc3dmIMMKNoeQJf20f5CK1Wl8cEV110RMRZ96uG9oOVbDneeM1n8U6-tIwxA5uUdskxRFRvLX0ep_tsPnuY80mhv1NcTZcByUK7HZcAc8R069DOHUZgxQHFIG0wxt nX7CZo7D79JBLYqz3k-Cg2sQ-AITylEFKHAUEcD-DaBNbd6QxSnjryuxN5SQgMaqVGxPBb5jUsz1iyApRE3TgbT8V1w", Jun 2023.

Table 13: Obfuscation Categories (Part 1)

Obfuscation Identifier	Obfuscation Name	Description
O1	Change Mapping of Variables	Use the same variable names, but switch the data that it applies to.
O2	Incorrect/Misleading Documentation	Write comments that are incorrect or misleading when describing the function of the code.
O3	Change variable names to imply different data type.	Example: std::string numberOfPeople; int name. New names may imply different function but focus was put on different data type.
O4	Insert Unused Variables	
O5	Insert Unneeded Print Statements	Have print statements that either, print variables that do not need to be printed or print information that is incorrect or irrelevant.
O6	Change For-Loops to Recursion.	
O7	Change Variables to Match a Functionality Completely Different	Example: A train booking system: int findTrain; int arrivalTime; int ticketPrice; bool bookTrain;
O8	Change Strings to use ASCII Characters	
O9	Represent Numbers as Unnecessary Math.	Instead of 10, write: (((((1+1+1)*3*5)+5)/10)+5)+(2/3). For integers, the (2/3) truncates to 0. There were instances that the LLM would not truncate.
O10	Insert Unnecessary If/Else Statements	Add if-statements that will have no effect on the outcome. For example, an if statement that always evaluates to true.
O11	Insert Unnecessary For-loops	This can be combined with O9 to do unnecessary math.
O12	Remove Spaces and New-lines	
O13	Combine O8,O9,and O11	Represent strings in their ASCII form. Represent ASCII characters through confusion math to get the ASCII decimal number you are needing, use loops to further confuse the math. At times, extra variables were used for conversions.
O14	Combine O9 and O11 with Unnecessary Variables	Use complex math, for-loops, and extra variables to represent numbers.
O15	Combine O2-O14	More or less combining every obfuscation method. Also using different languages and alphabets. Excludes O6. Not all obfuscation types may be shown in a given obfuscation. The goal is to try and combine all methods up to this point but not explicitly. When obfuscating, typically started with O13 and when backwards apply obfuscations. O5 was not always used.
O16	Change Output Presentation	Change the way the output is presented. For Example: "123" instead of "321" or "[231]"
O17	Combine O2, O7, O14, and O16	Aimed at making the code function appear different, without having messy code. The comments and variables should match. The for-loops and math can be used to add fake functionality. Changing the output to completely change the look of the info. The goal is to keep it clean, so as to make sure it does not think that it is an obfuscation. Focus was put on the variable names and the comments.
O18		This obfuscation could be categorized as a mix of in-congruent commenting, variable renaming, character replacement, and redundant or misleading operations. It is an attempt at making the code harder to understand at a glance.

Table 14: Base Code

Name	Description
B1	Print the integer numbers 1 to 10, each on a newline.
B2	Print the sum of the numbers 1 to 10.
B3	Print 10 factorial.
B4	Find the factors of 10 and print them on the same line separated by a space.
B5	Concatenate the two strings "race" and "car" together and print the result.
B6	Print the string "hello" 10 times, one line, no spaces.
B7	Concatenate four "Hello" and seven "There" together and print the result.
B8	Print the number of odd numbers in a vector of integers.
B9	Print the odd numbers from a vector of integers on one line separated by a space.
B10	Print the number of vowels in the string "alphabet".
B45	Swap the values of two variables. Print the before and after values of the variables.
B46	Check whether an inputted character is a vowel, upper or lower case. Print true or false.
B47	Find the compound interest for a principal of 10,000, rate of 5 and time of 2. Print the result of the calculation.
B48	Calculate the result of taking two numbers as input and raising one to the power of the other.
B49	Calculate and print the product of two inputted numbers.
B50	Check whether or not the inputted integer is prime. Print "true" or "false".
B51	Print these four strings on separate lines: "adsf!fjelnbo./23@#45jalkd", "as;lkdjfoine!,djfoekngrn", "apple!a;lkdjfoie", "This is the fourth line!"
B52	Four lines are printed. Each line contains an exclamation point proceeded and followed by a random amount,0-10, of random lowercase letters.
B53	Checks if a word is a palindrome, excluding the letter 'x'. Prints "true" or "false".
B54	Takes the inputted word and changes the middle character with '>'. Prints the result.
B55	Prints a space followed by 6 newlines.
	(Base codes B11 through B46 were created but not used due to time constraints)

Figure 5: PaLM: Base Code

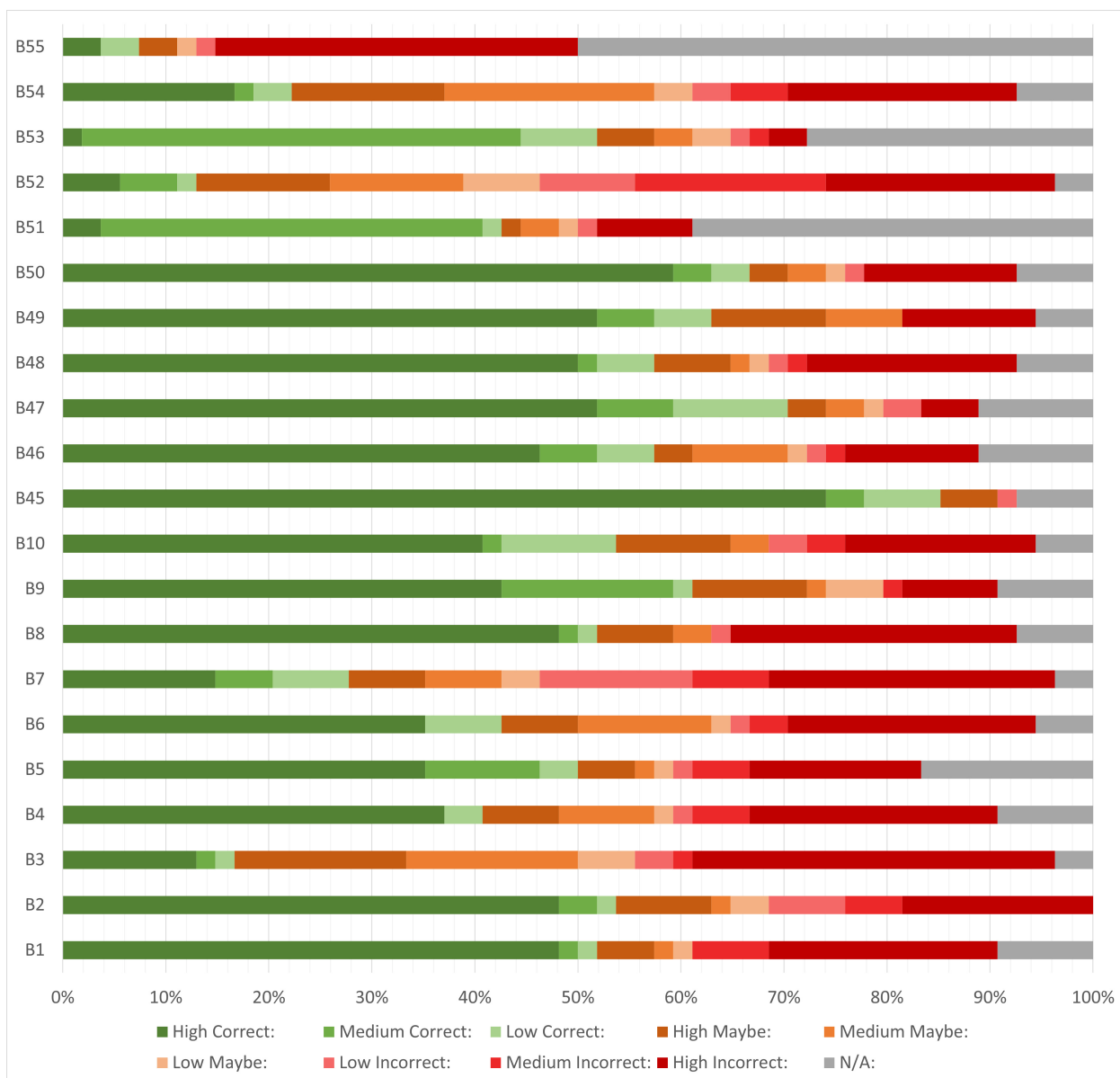


Figure 6: PaLM: Obfuscation

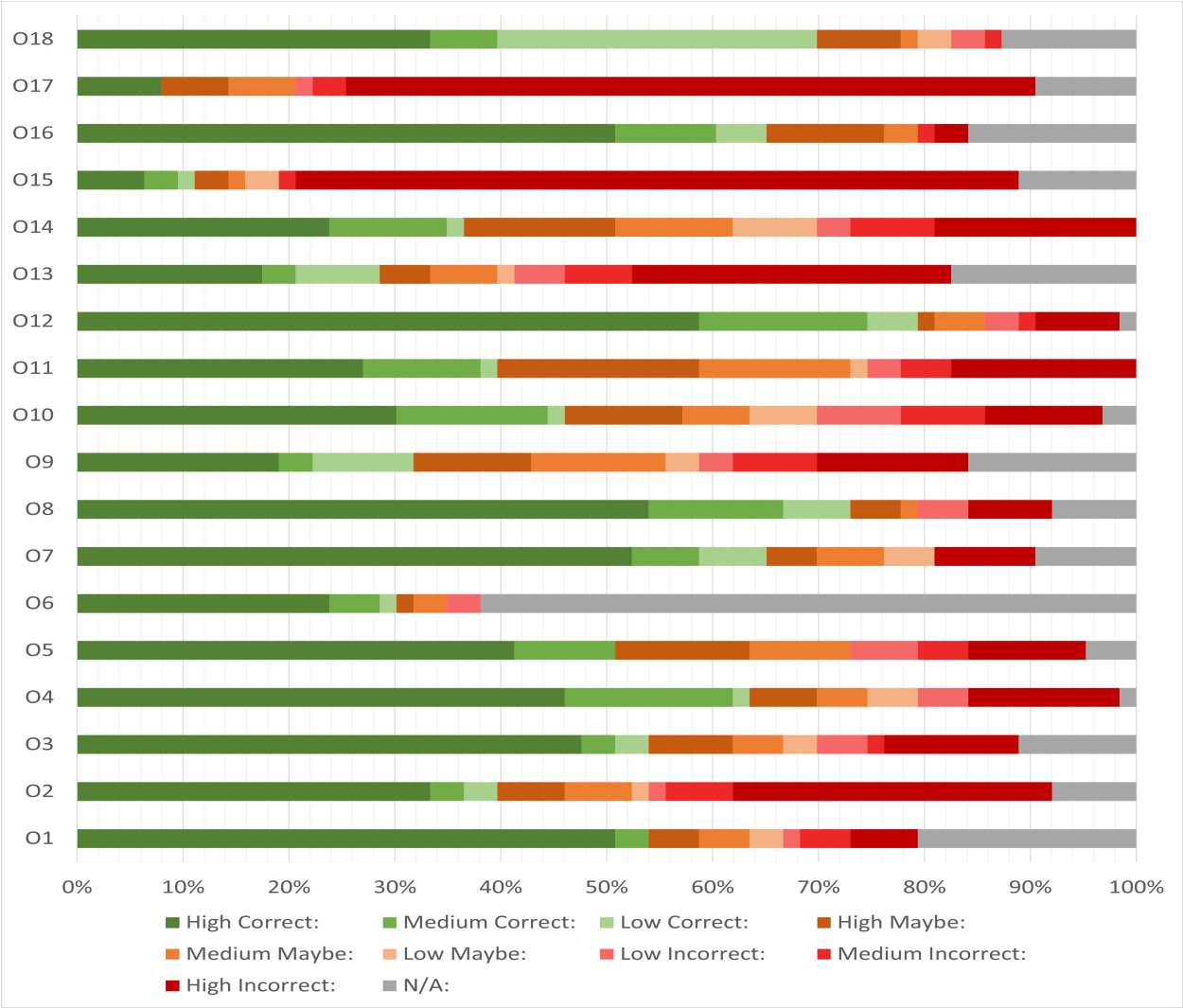


Figure 7: Base Code Ranking: Incorrect

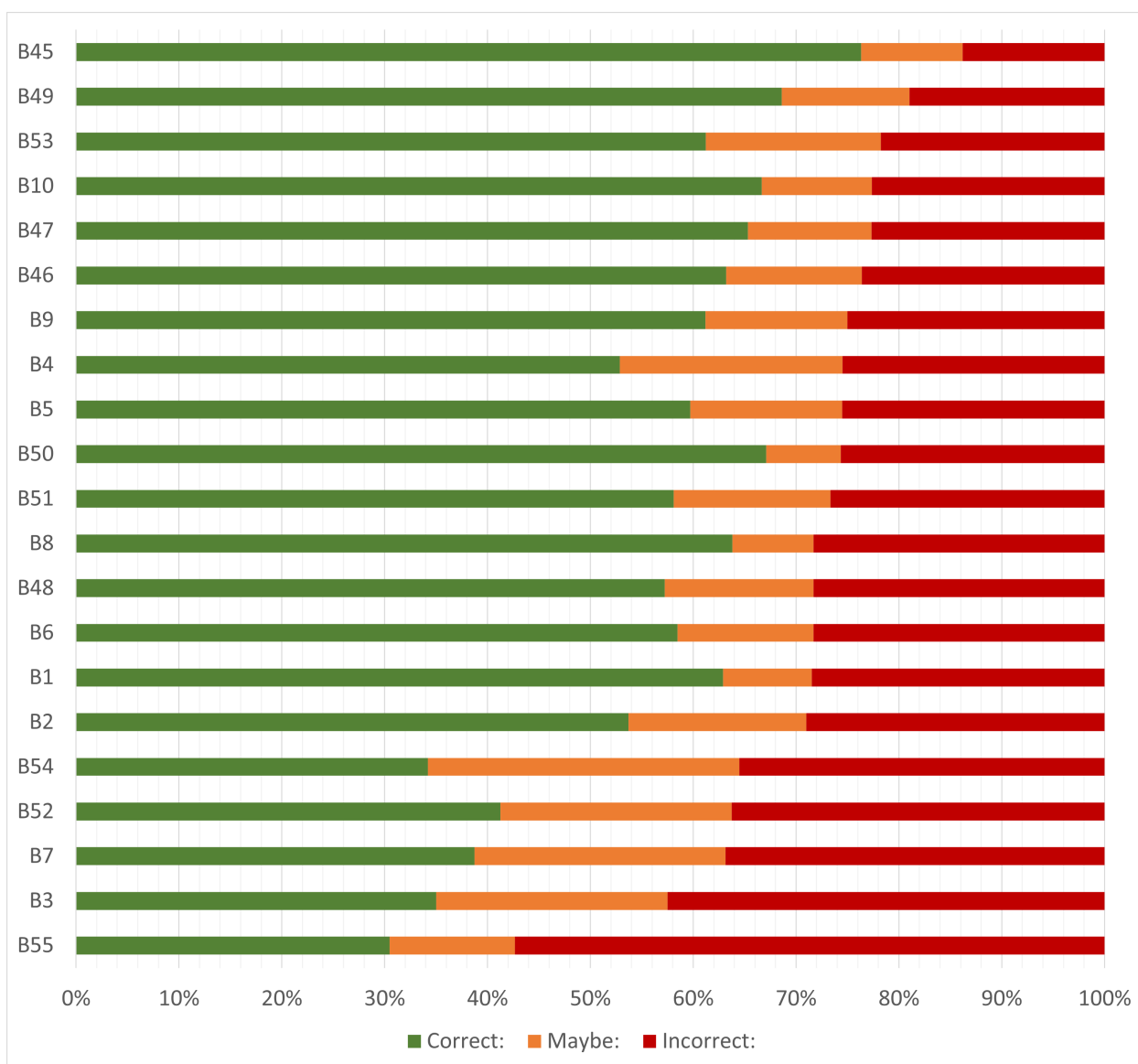


Figure 8: Obfuscation Ranking: Incorrect

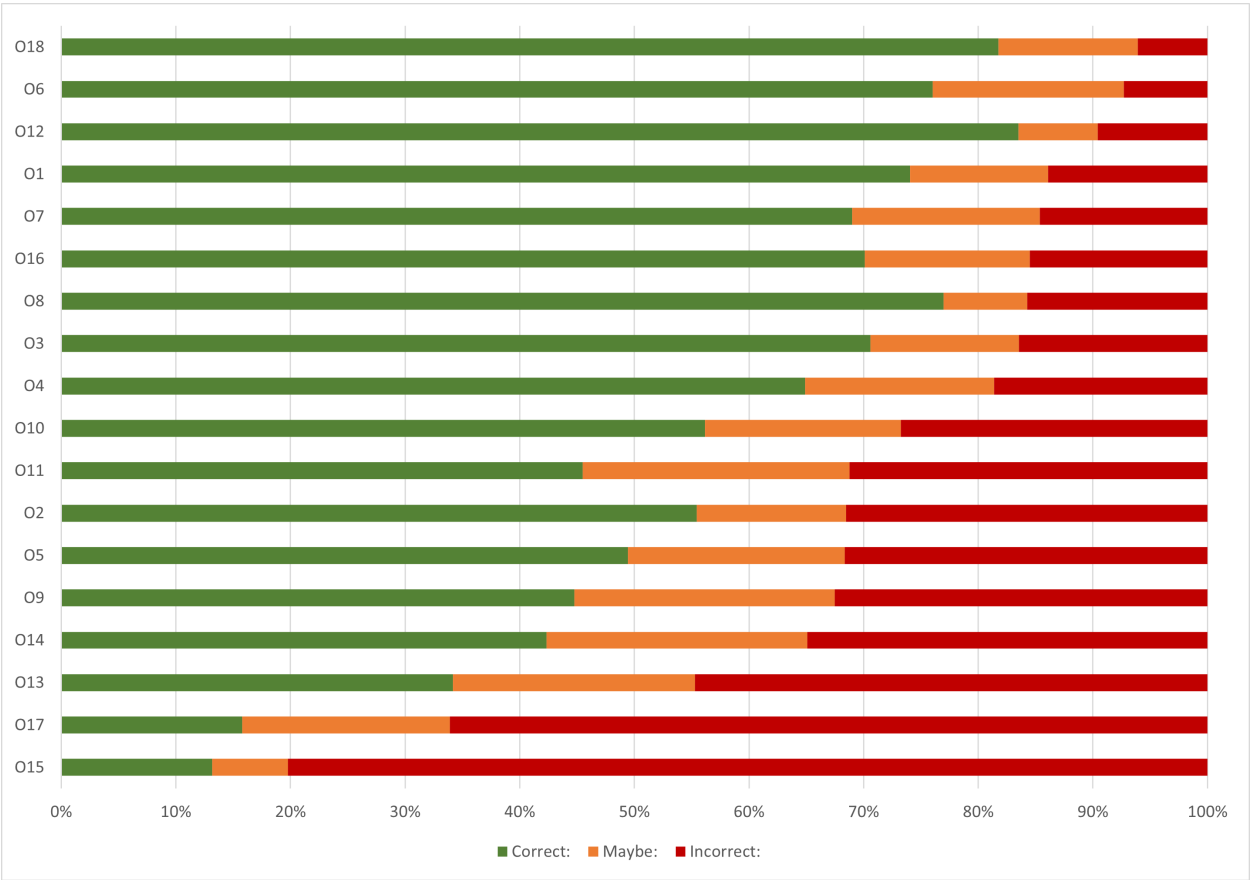


Figure 9: Prompts Results

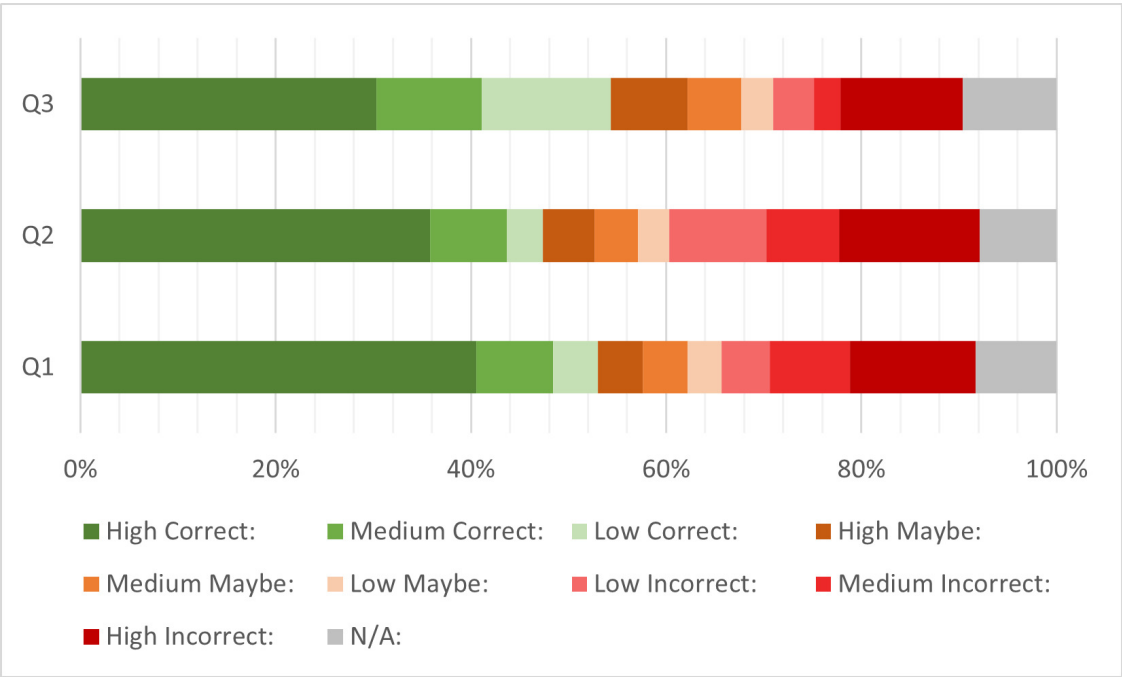


Figure 10: LLM v. LLM

